Event Modeling

From Sticky Notes to Go

Agenda

- Event Storming
- CQRS + Event Sourcing
- Code Examples

Event Storming

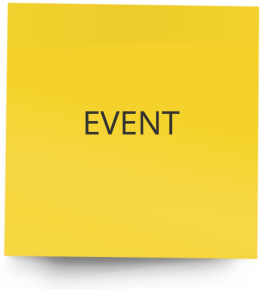
A workshop-based method to explore a domain

Historical Fact

Imperative Action

Policy/Rule applied to previous events

Legend

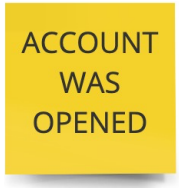


Historical Fact

Imperative Action

Policy/Rule applied to previous events

Legend



Bank Account Example

ACCOUNT WAS OPENED

MONEY WAS DEPOSITED

Bank Account Example

Bank Account Example

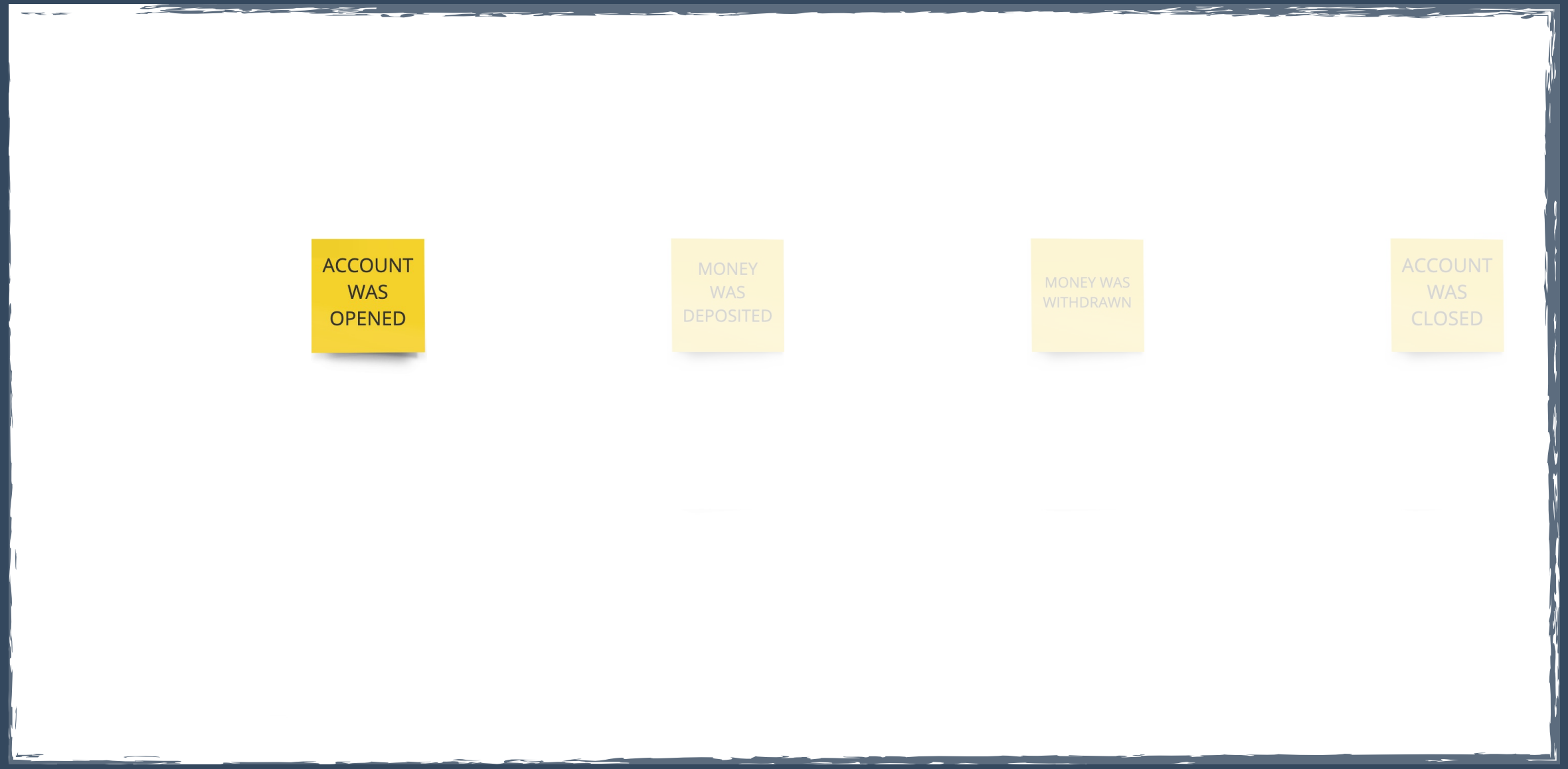
Bank Account Example

Historical Fact

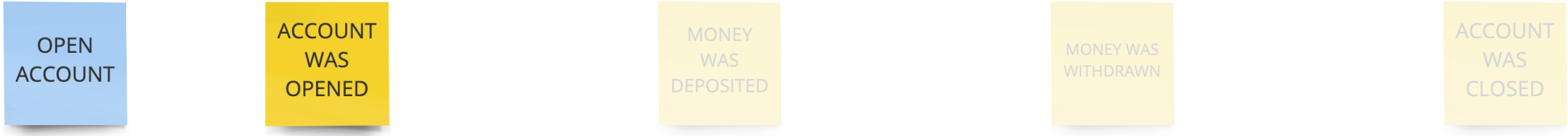
Imperative Action

Policy/Rule applied to previous events

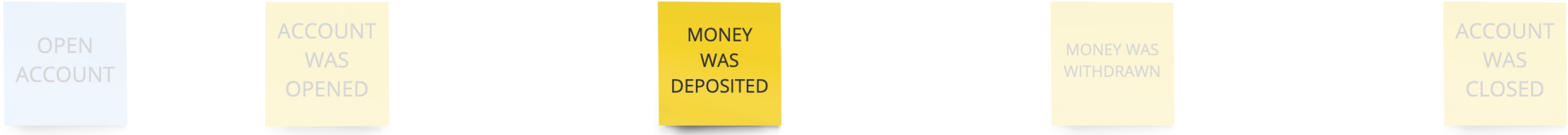
Legend



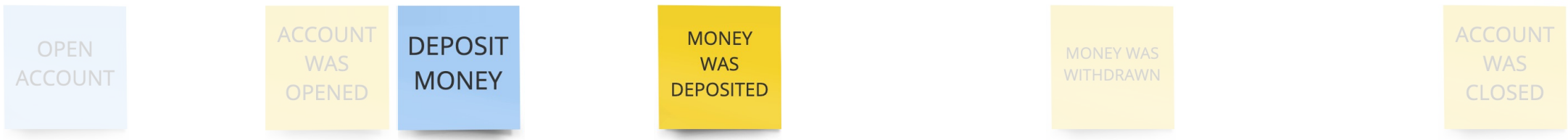
Bank Account Example



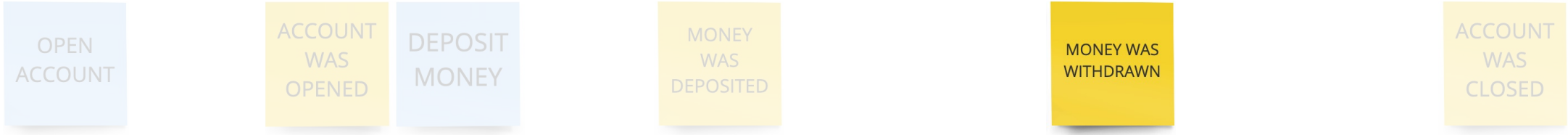
Bank Account Example



Bank Account Example

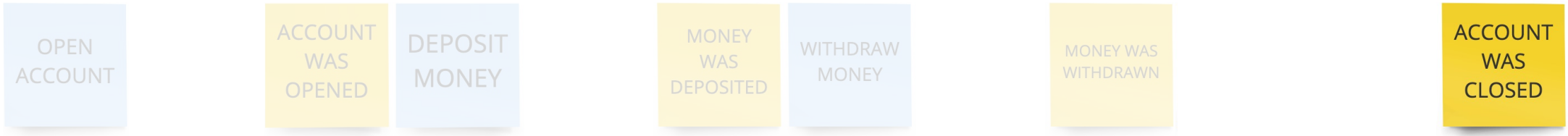


Bank Account Example

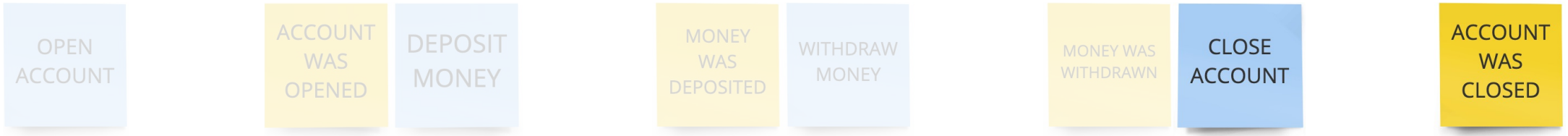


Bank Account Example

Bank Account Example



Bank Account Example



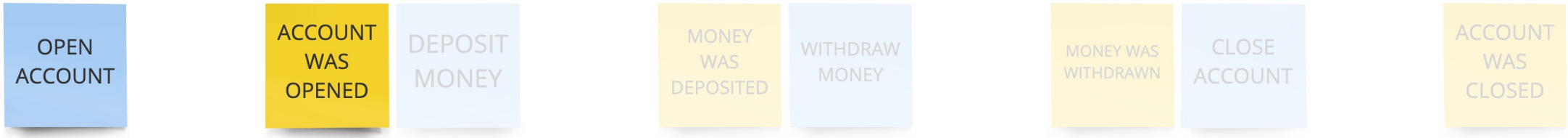
Bank Account Example

Historical Fact

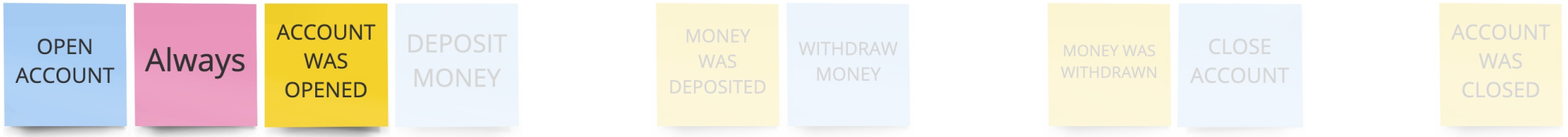
Imperative Action

Policy/Rule applied to previous events

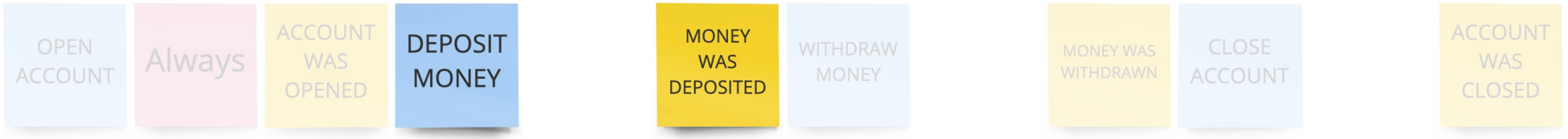
Legend



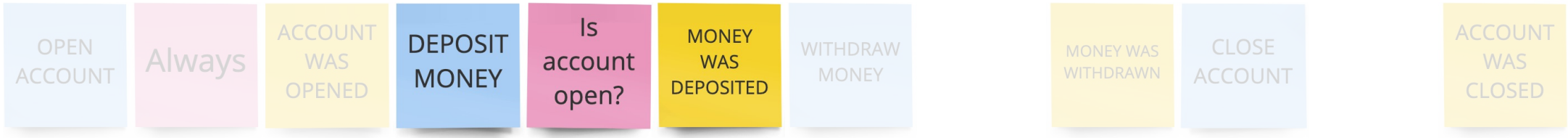
Bank Account Example



Bank Account Example



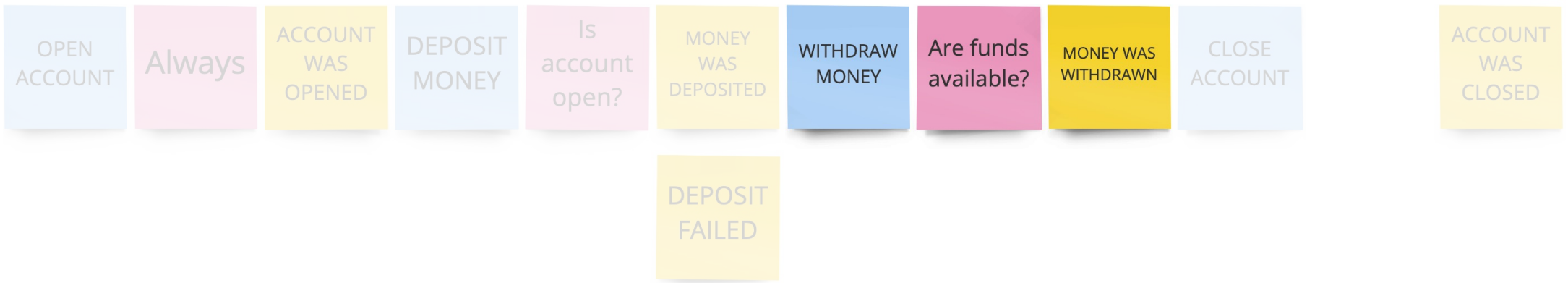
Bank Account Example



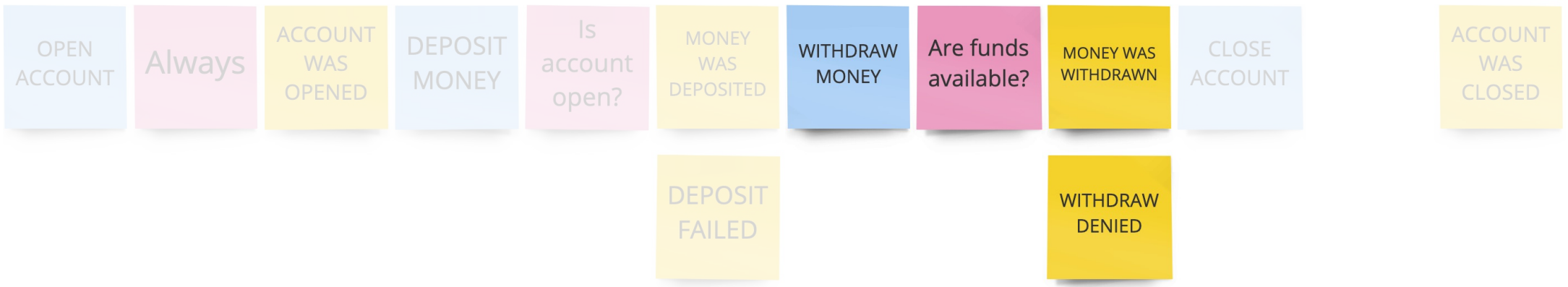
Bank Account Example

Bank Account Example

Bank Account Example



Bank Account Example



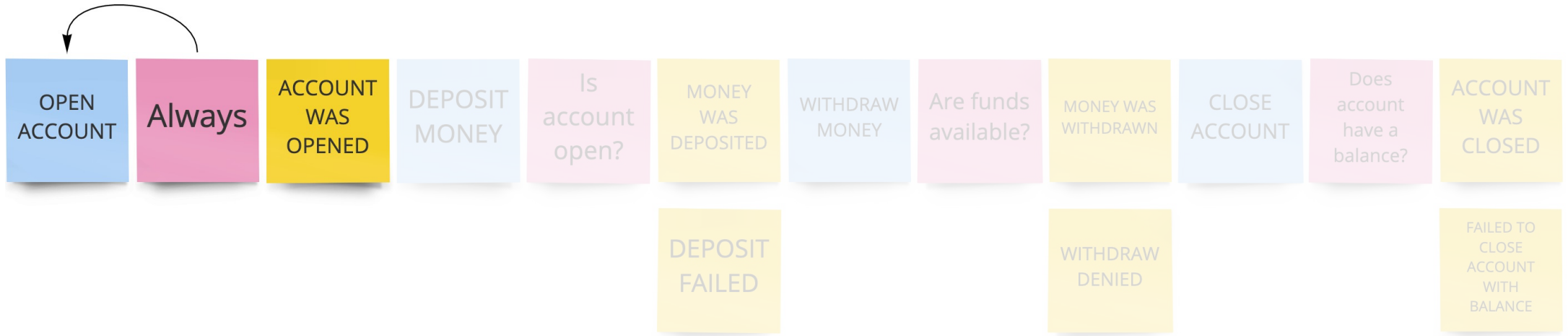
Bank Account Example

Bank Account Example

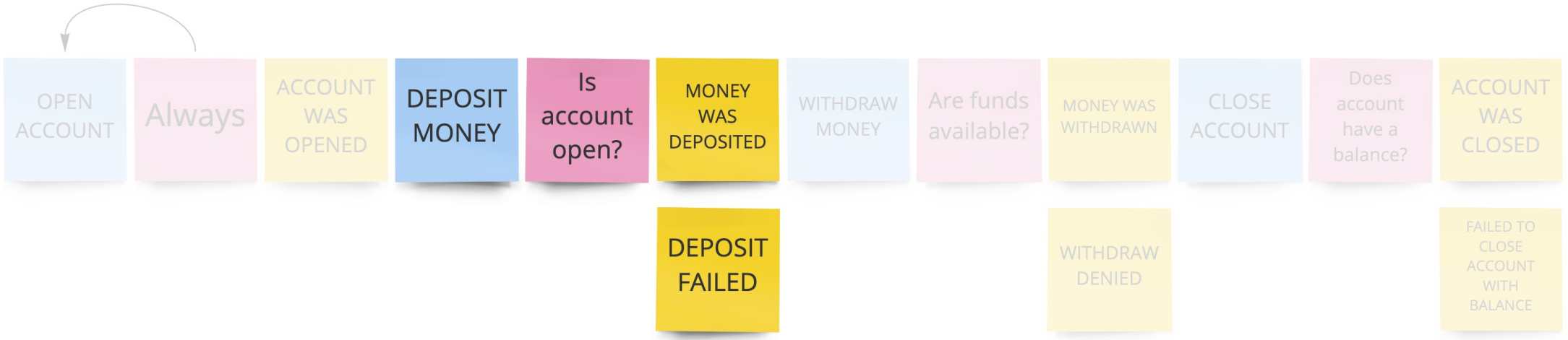
Bank Account Example

Bank Account Example

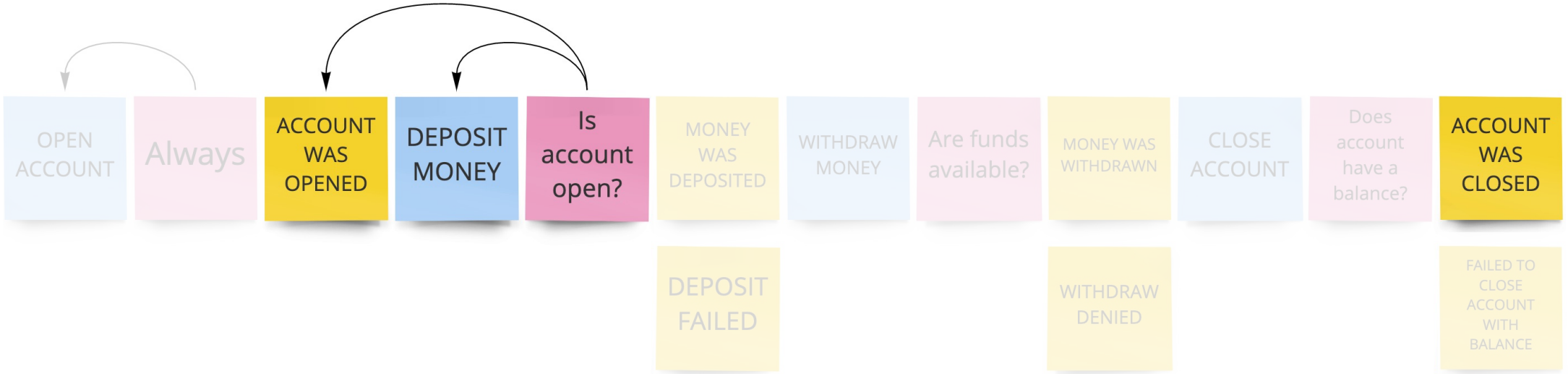
Bank Account Example



Bank Account Example

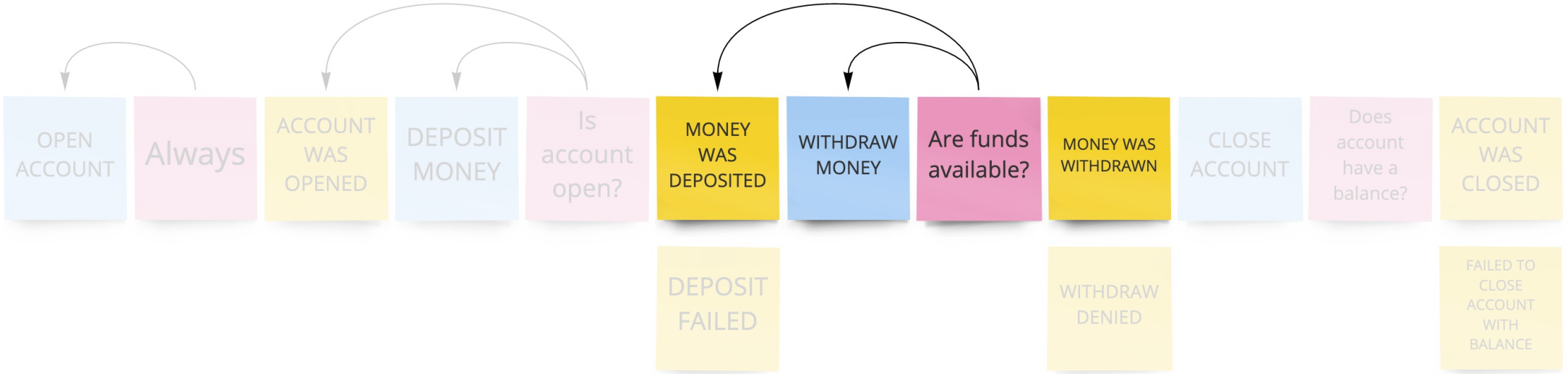


Bank Account Example

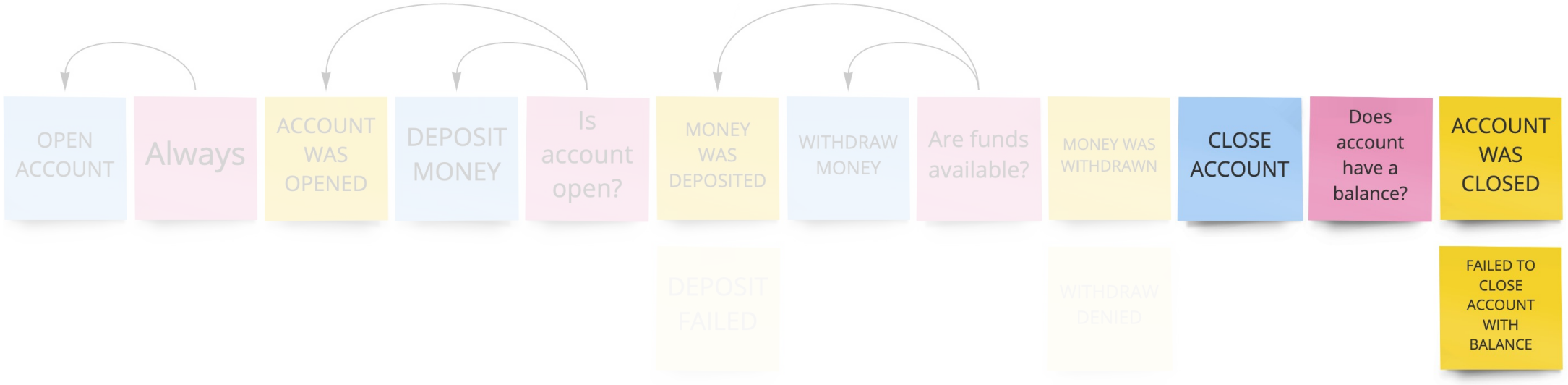


Bank Account Example

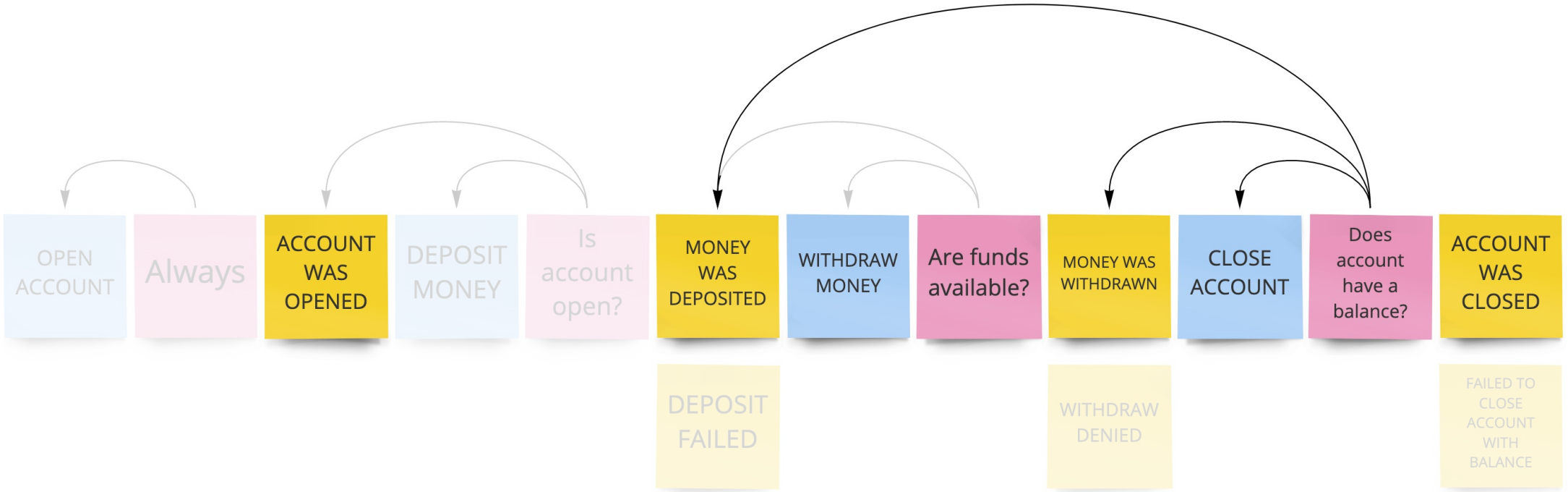
Bank Account Example



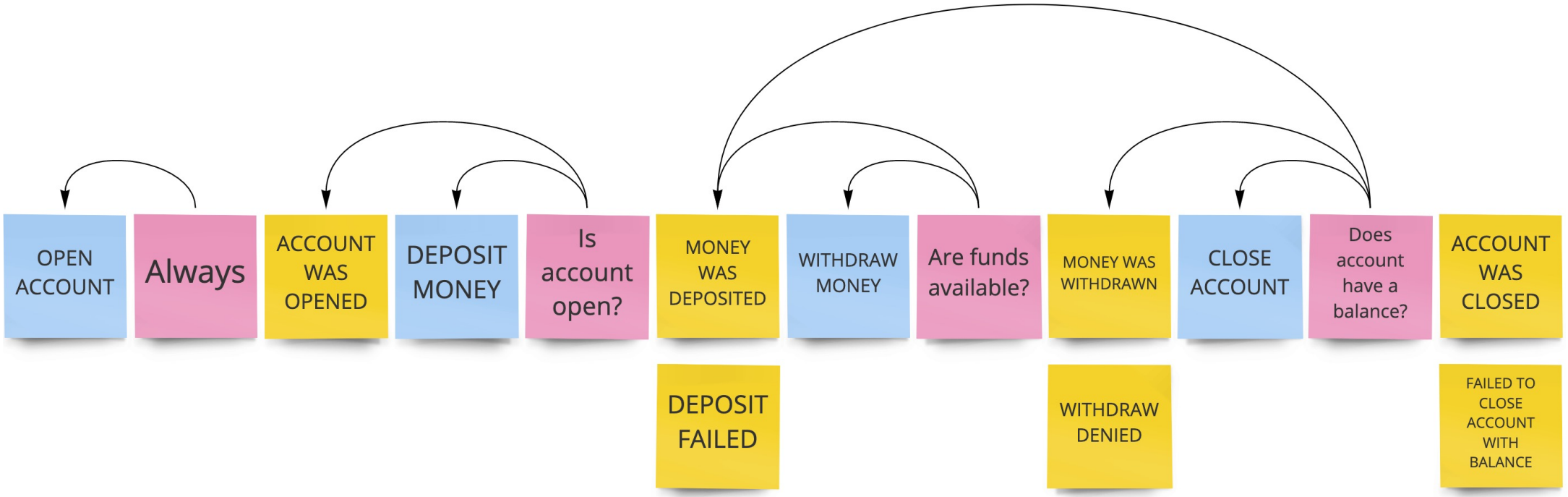
Bank Account Example



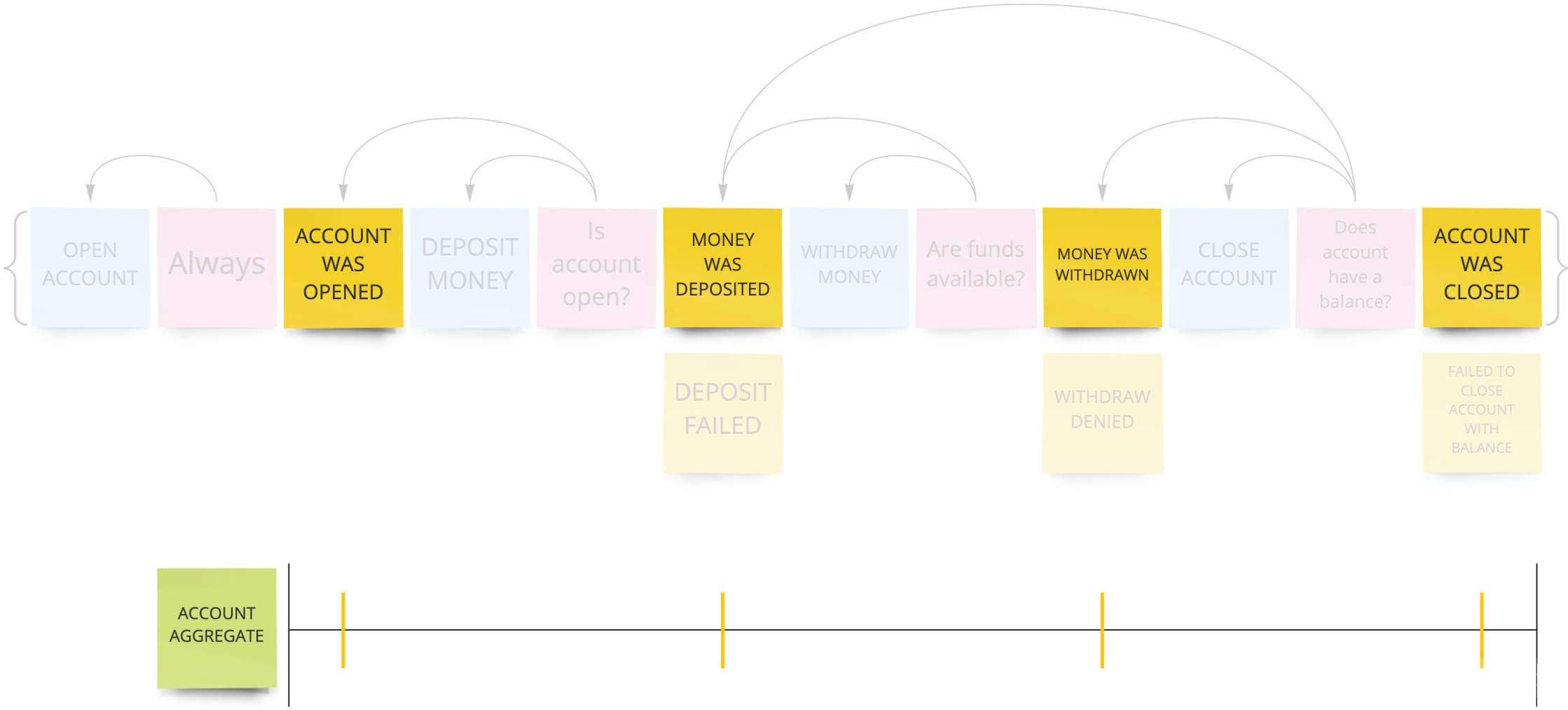
Bank Account Example



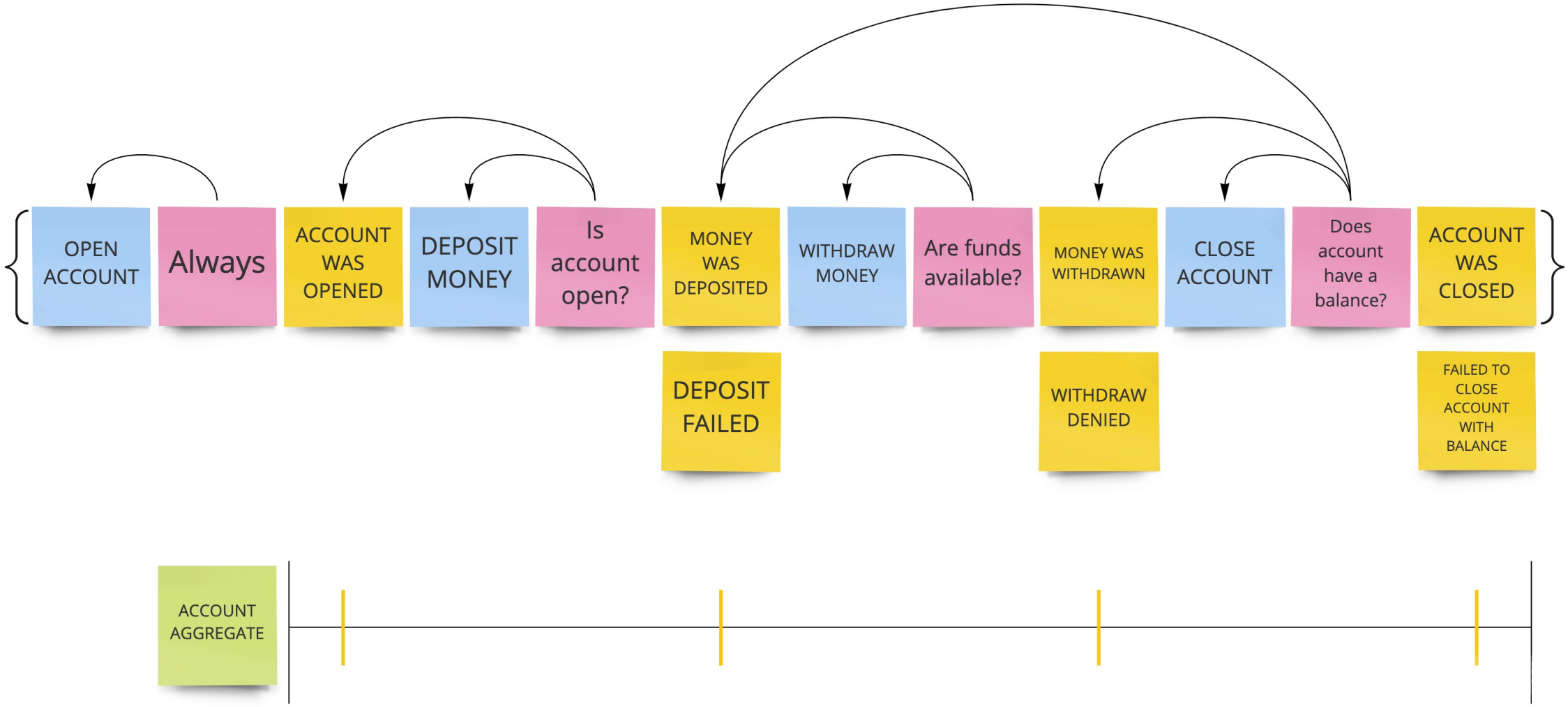
Bank Account Example



Bank Account Example



Bank Account Example



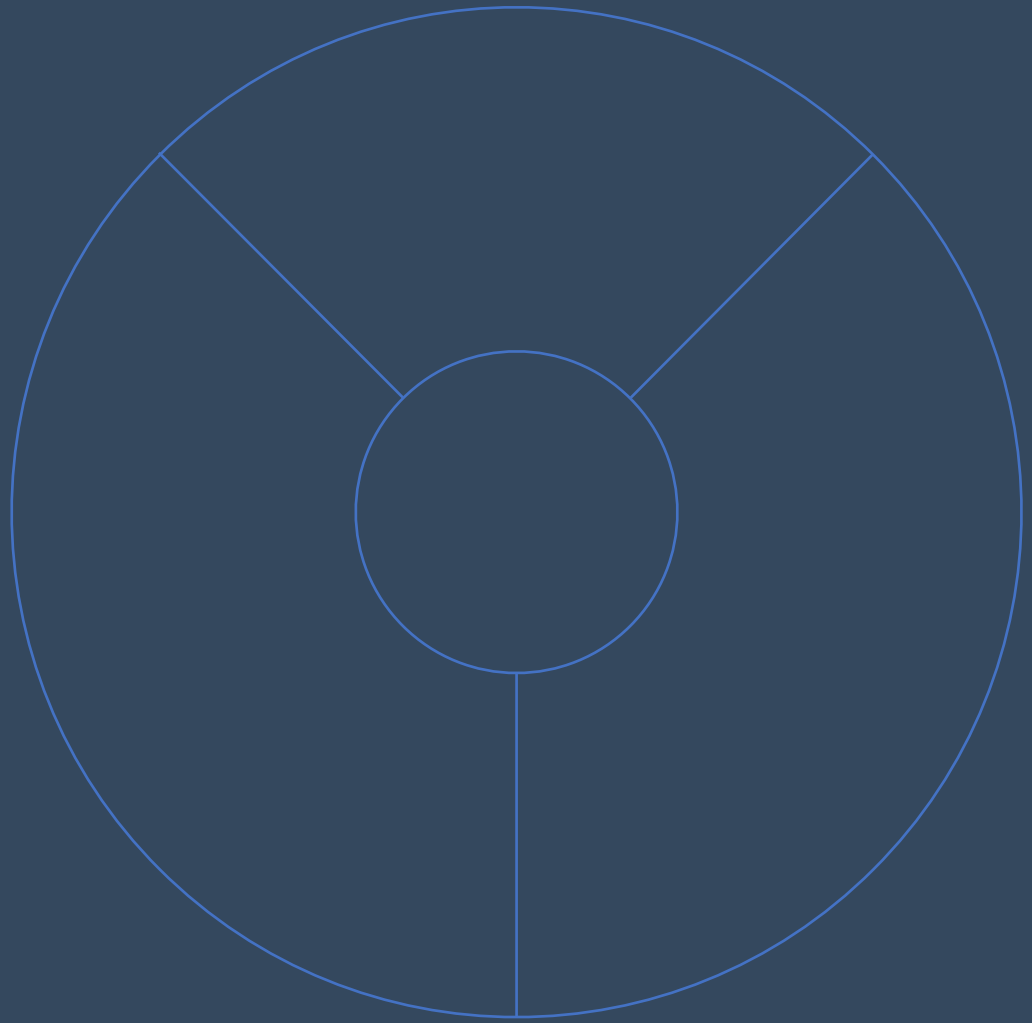
Bank Account Example

CQRS + Event Sourcing

Command Query Responsibility Segregation

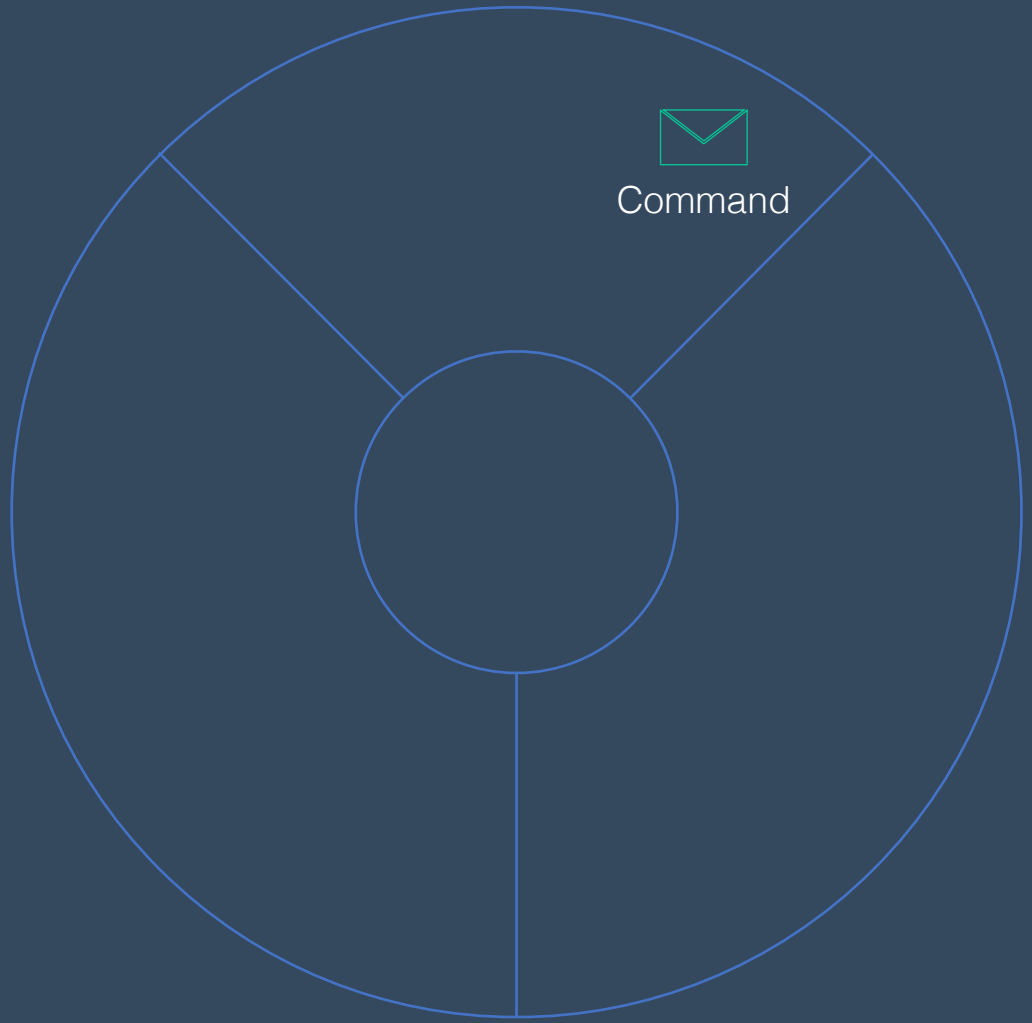
CQRS-ES

API



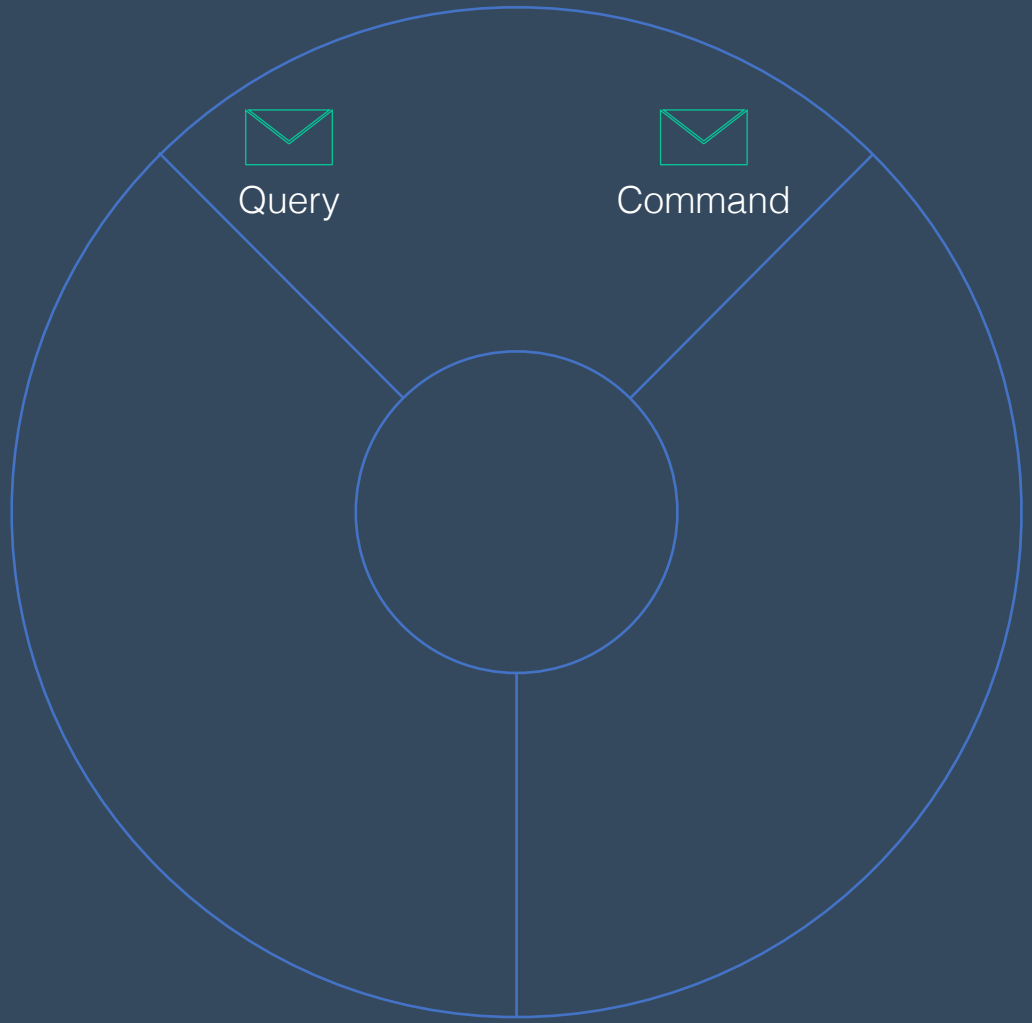
CQRS-ES

API



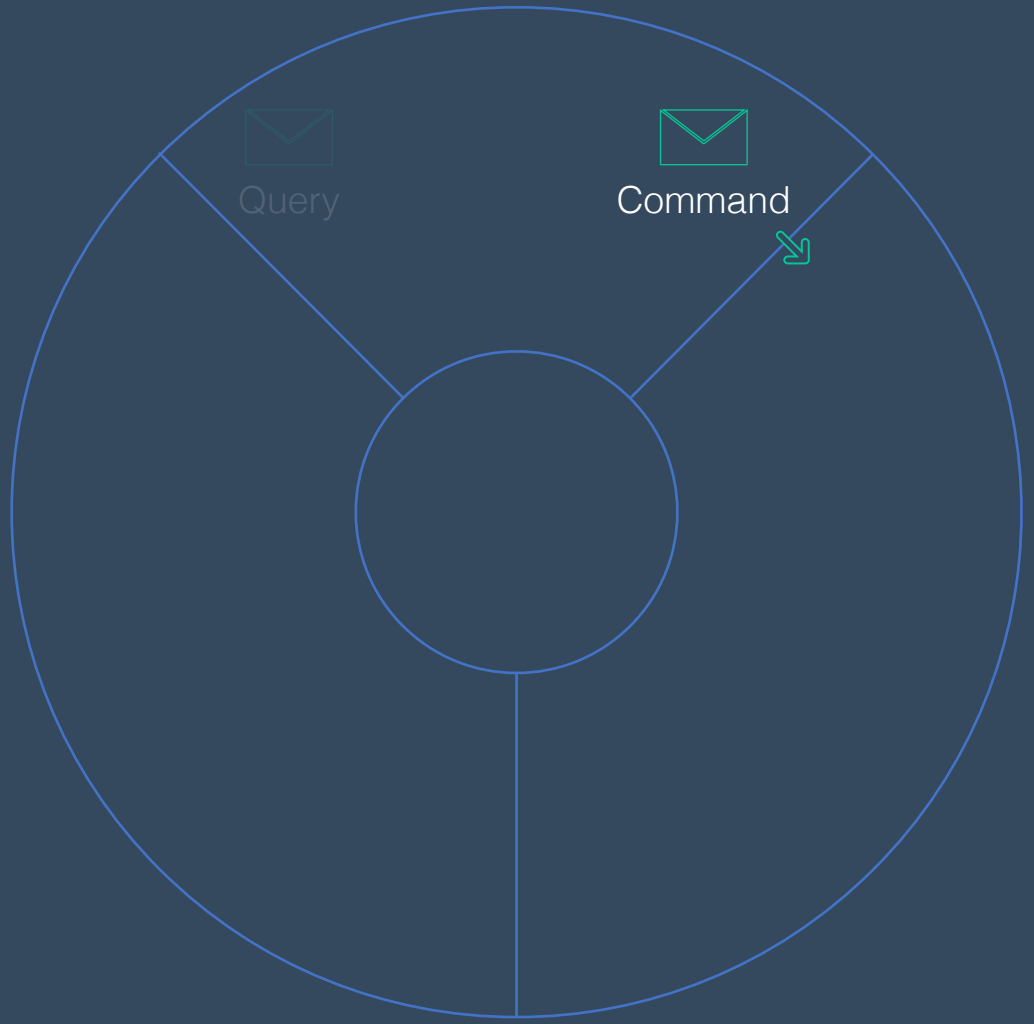
CQRS-ES

API



CQRS-ES

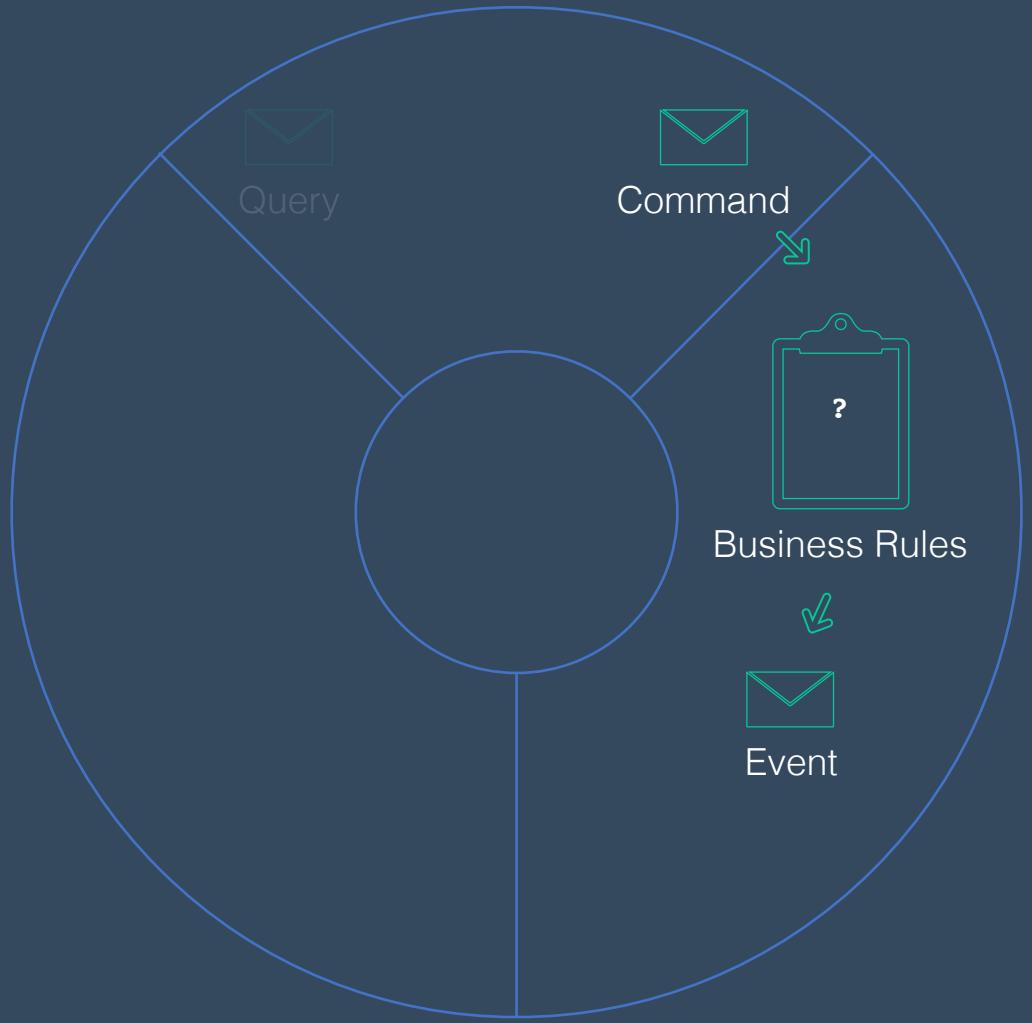
API



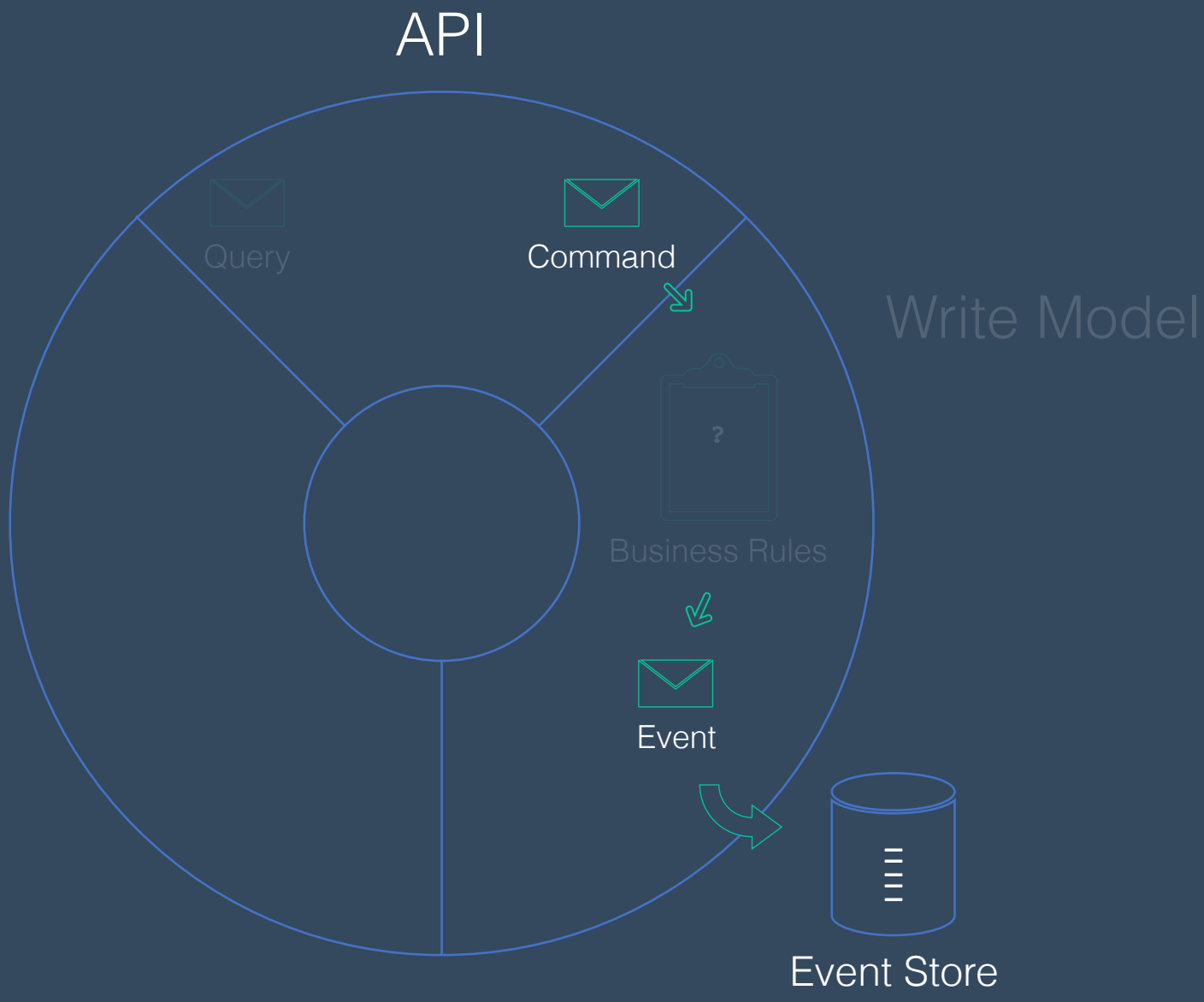
Write Model

CQRS-ES

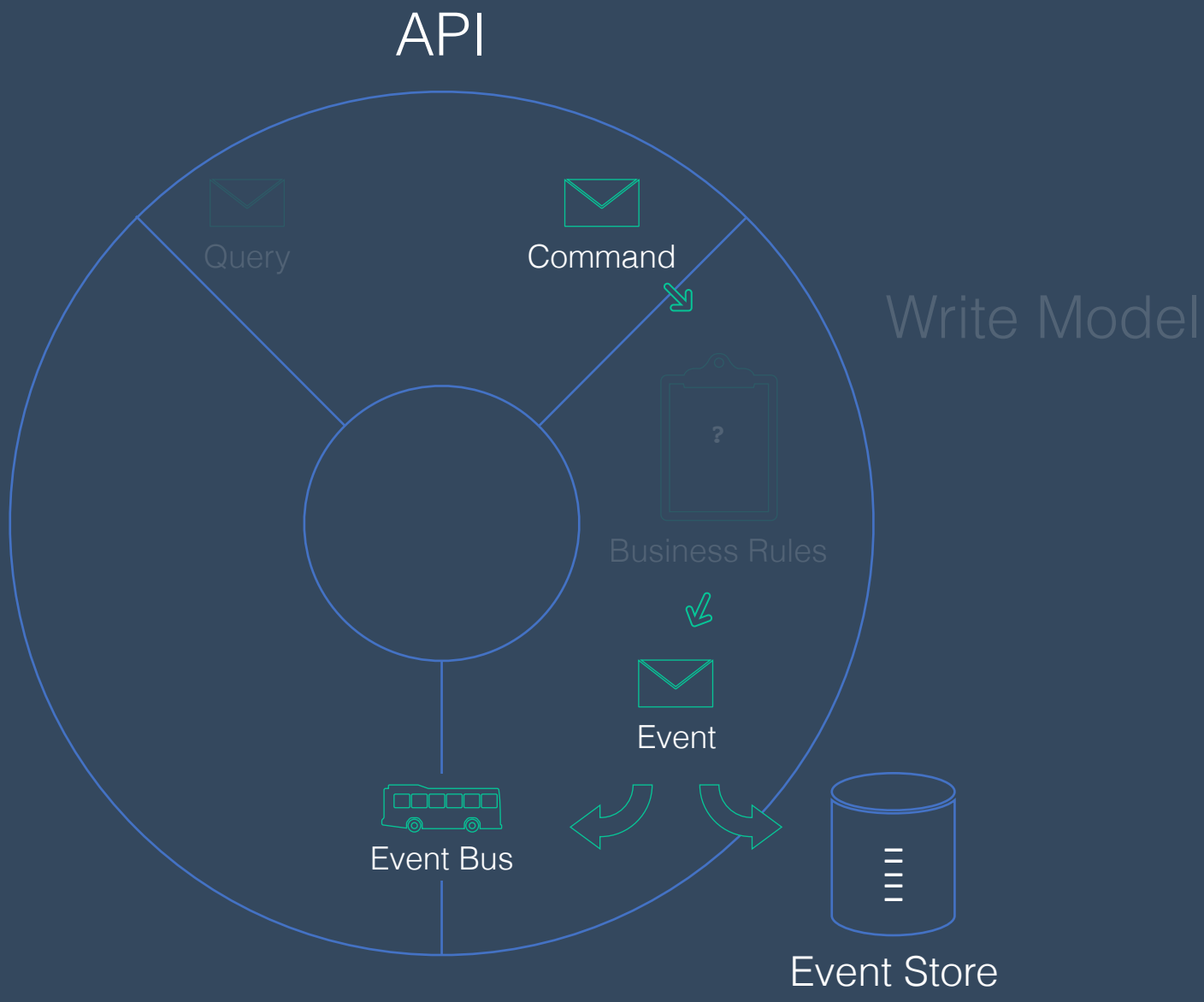
API



CQRS-ES

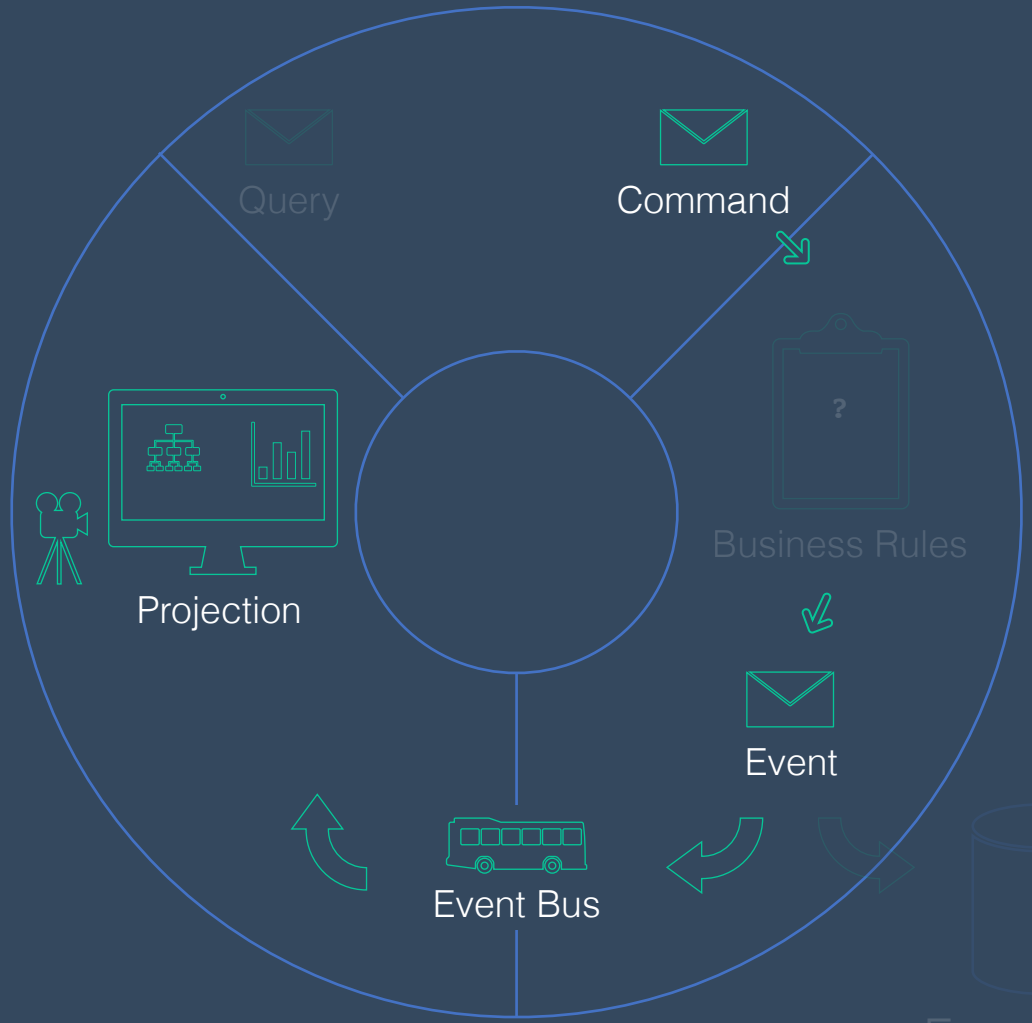

API
Query
Command
Write Model
?
Business Rules
Event
Event Store

CQRS-ES


API
Query
Command
Write Model
?
Business Rules
Event
Event Bus
Event Store

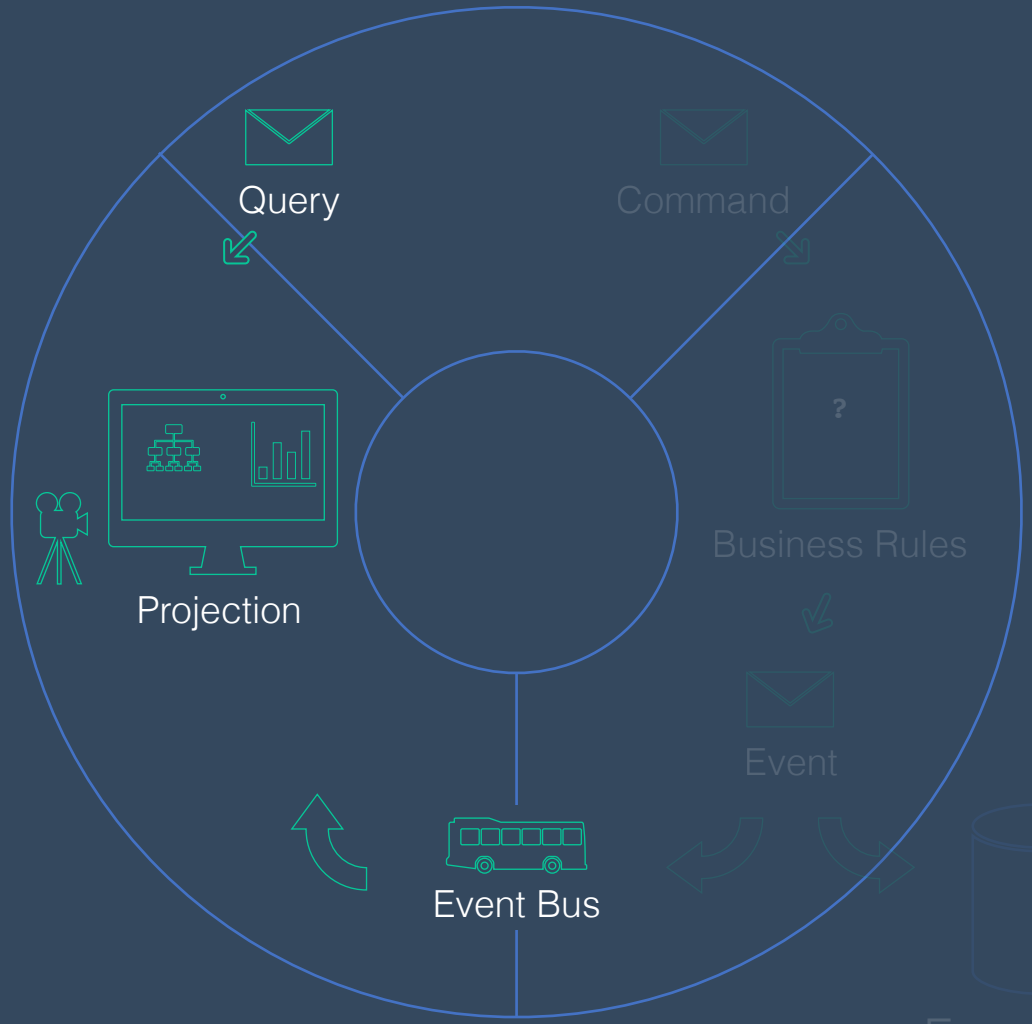
CQRS-ES

API


Query
Command
Read Model
Write Model
Projection
Business Rules
Event
Event Bus
Event Store

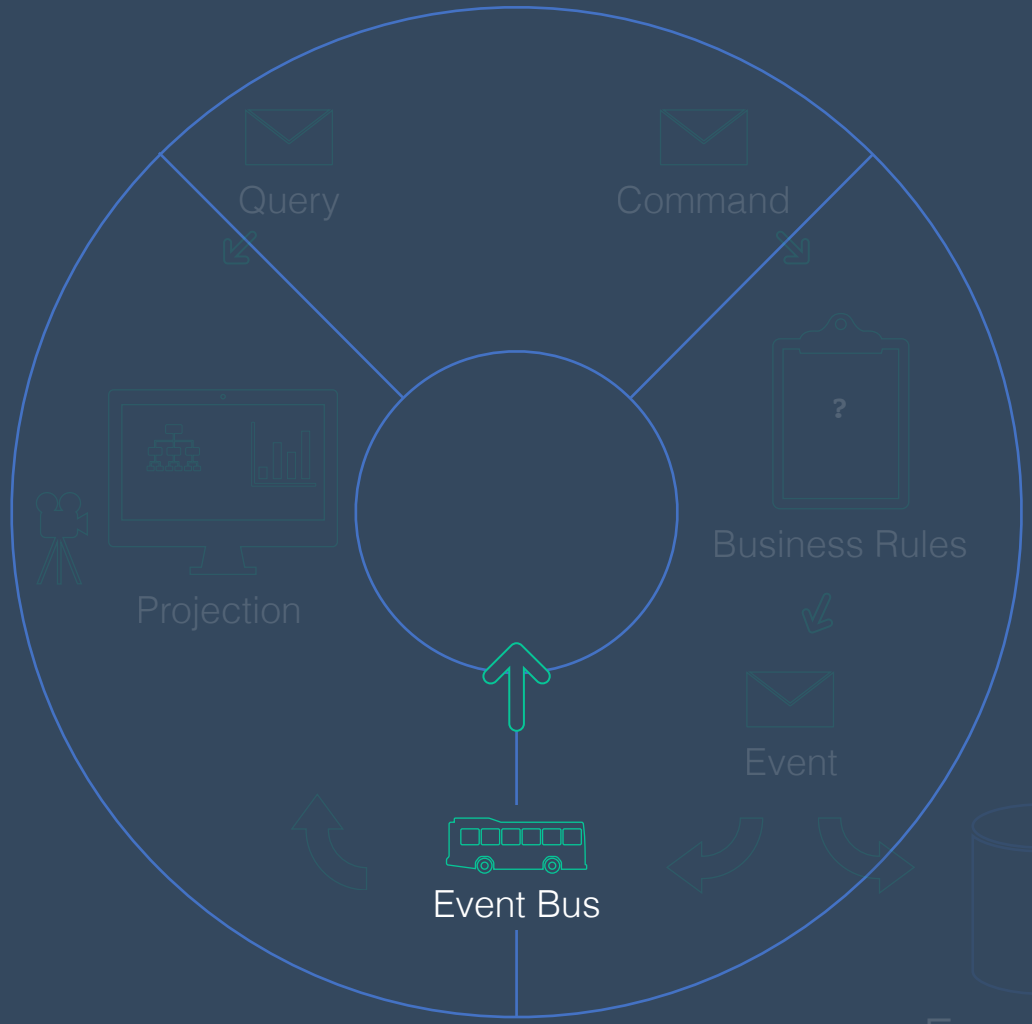
CQRS-ES

API


Query
Command
Read Model
Write Model
Projection
Business Rules
Event
Event Bus
Event Store

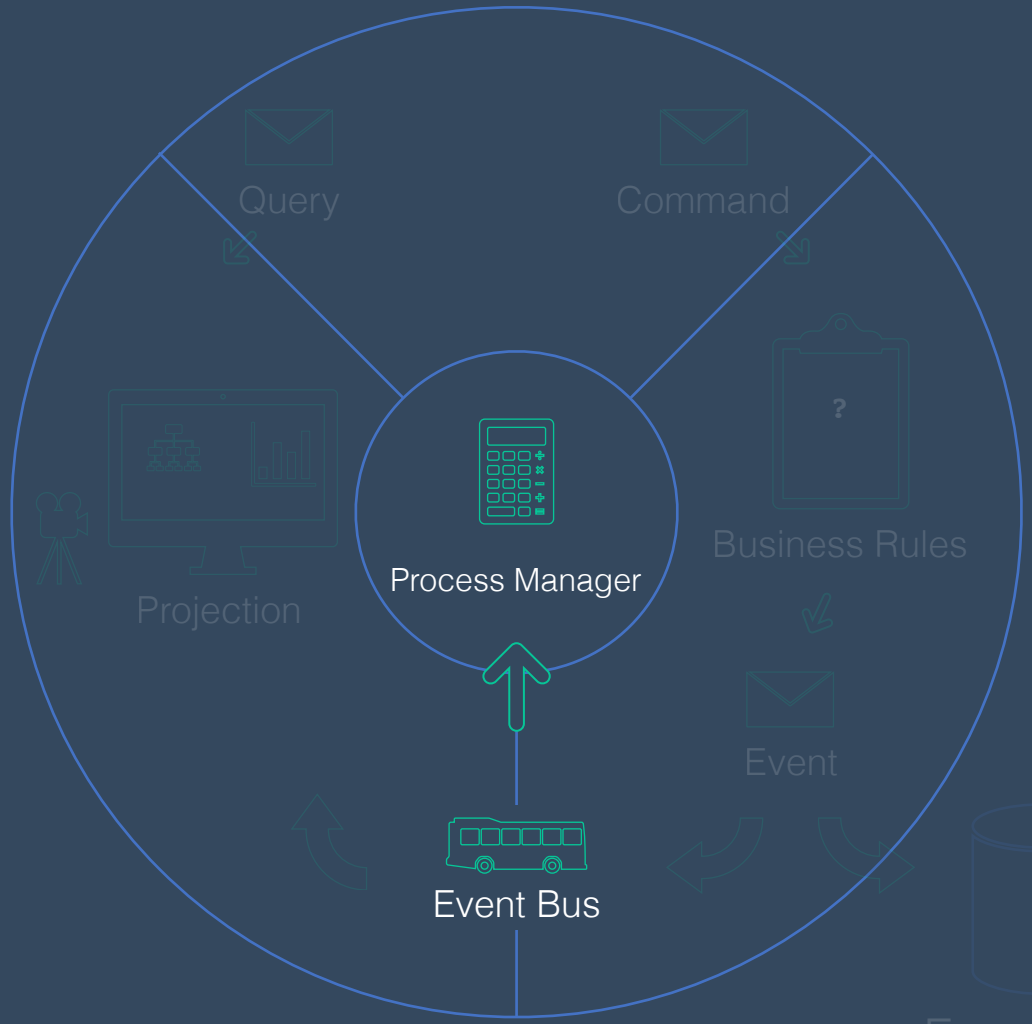
CQRS-ES

API

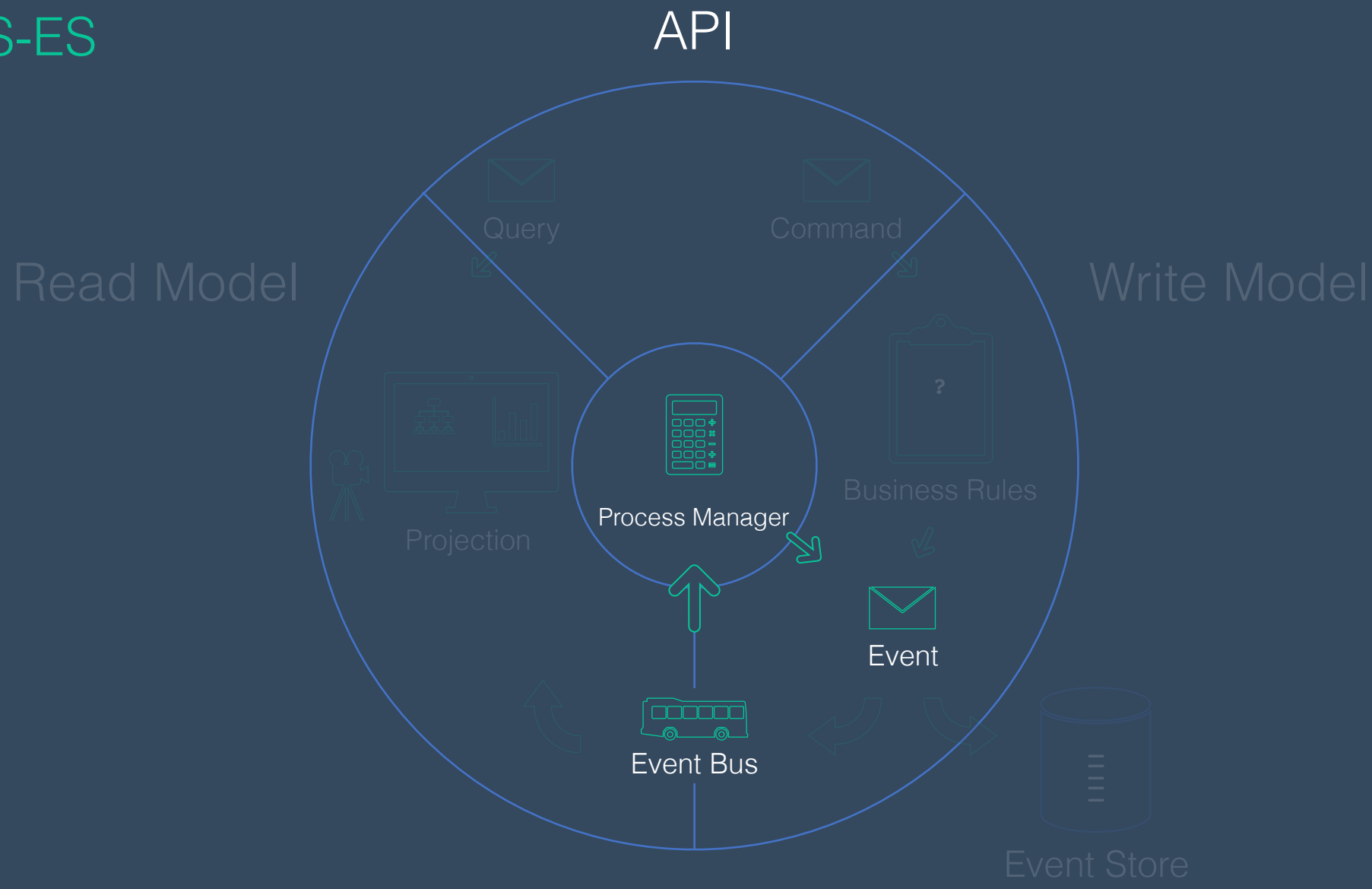

Query
Command
Read Model
Write Model
Projection
Business Rules
Event
Event Bus
Event Store

CQRS-ES

API

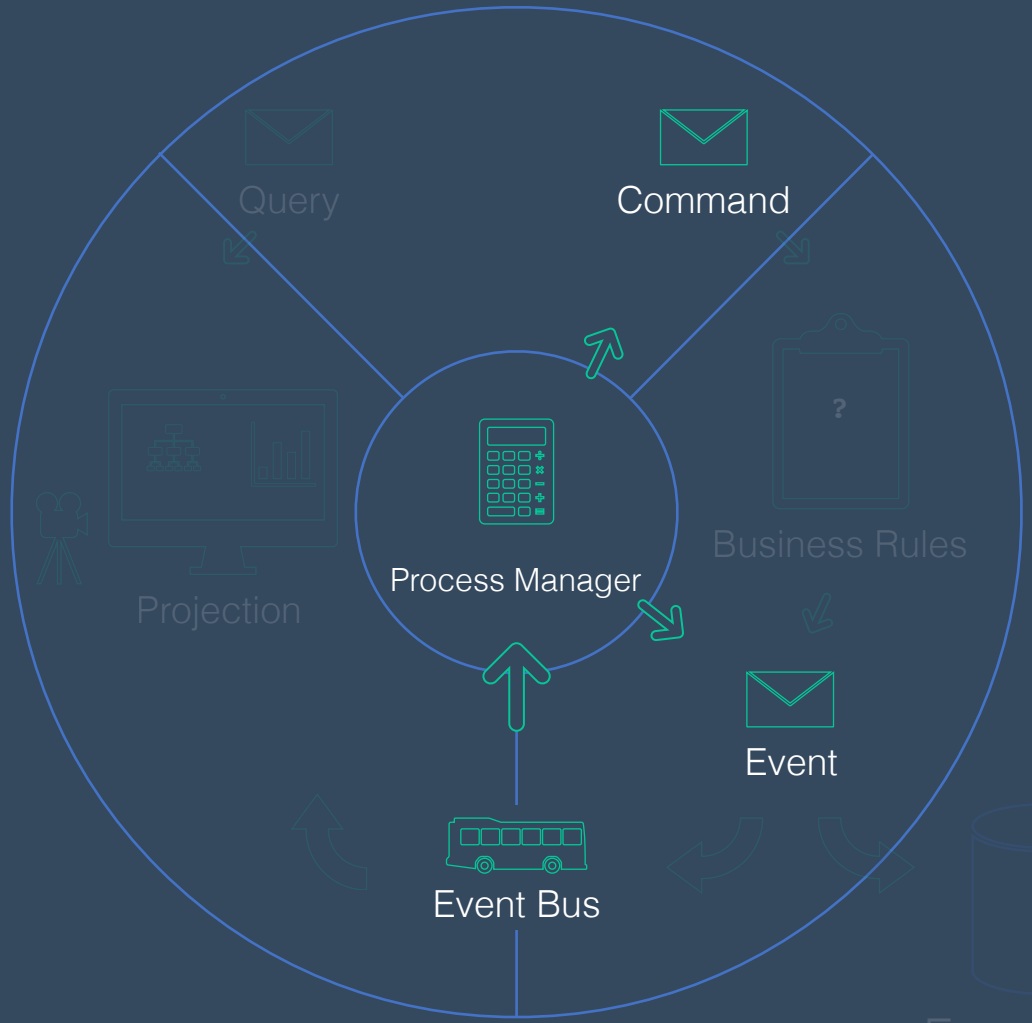

Query
Command
Read Model
Write Model
Projection
Process Manager
Business Rules
Event
Event Bus
Event Store

CQRS-ES


API
Read Model
Write Model
Query
Command
Projection
Process Manager
Business Rules
Event
Event Bus
Event Store

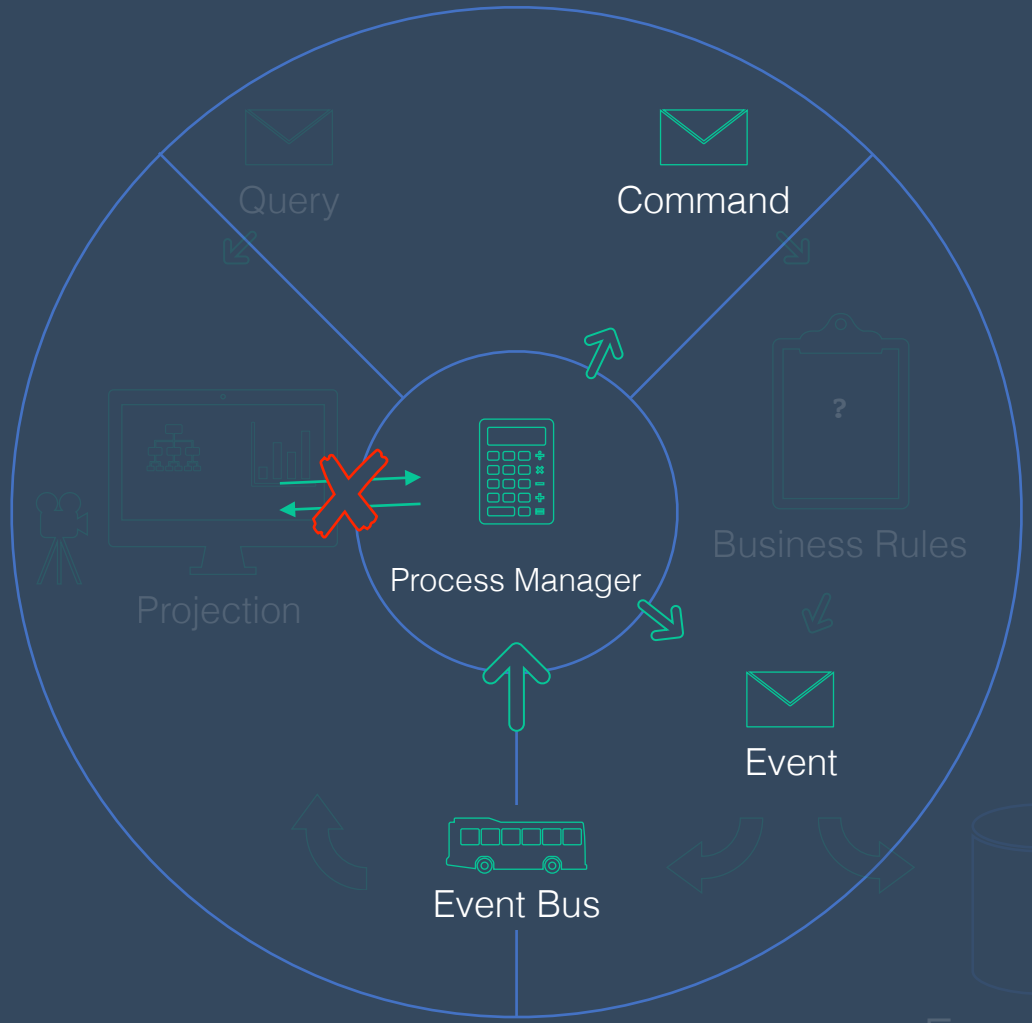
CQRS-ES

API


Query
Command
Read Model
Write Model
Projection
Process Manager
Business Rules
Event
Event Bus
Event Store

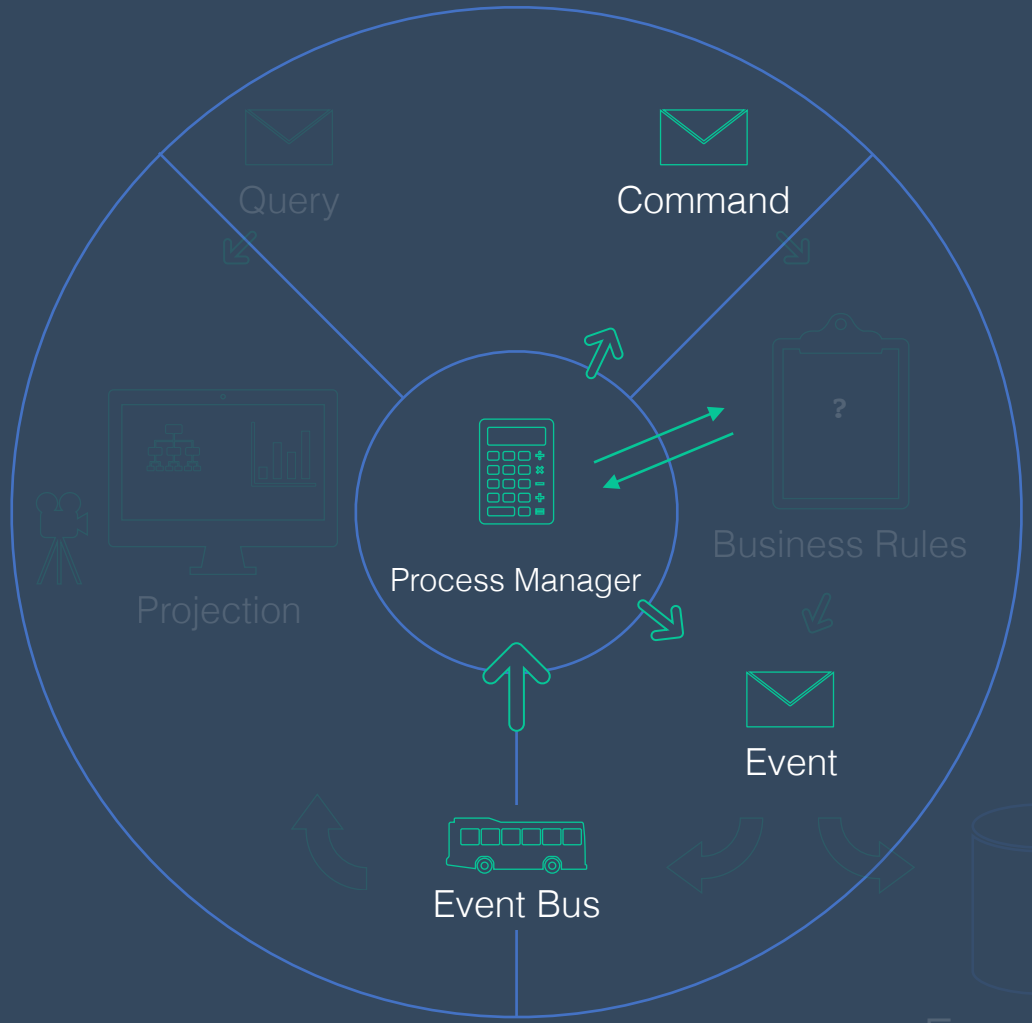
CQRS-ES

API

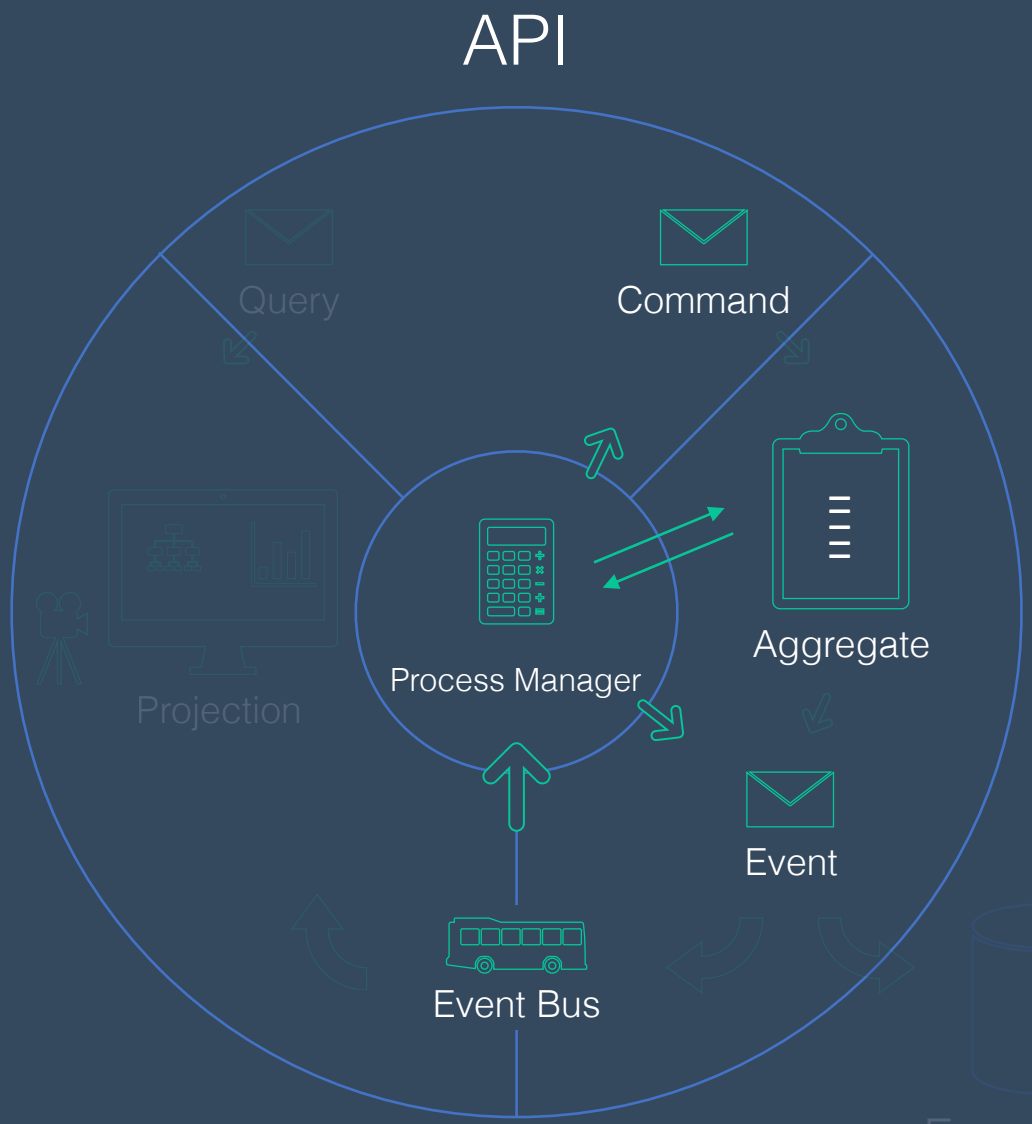

Query
Command
Read Model
Write Model
Projection
Process Manager
Business Rules
Event
Event Bus
Event Store

CQRS-ES

API

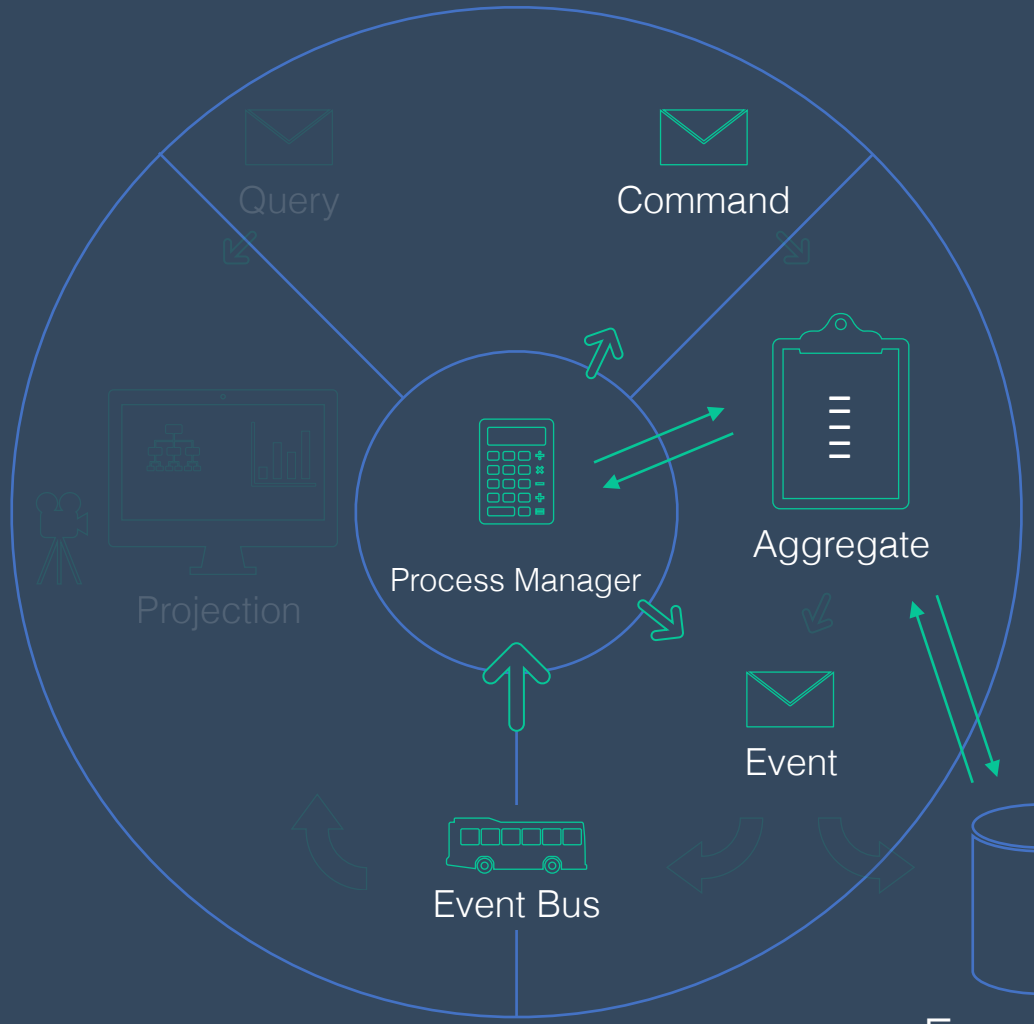

Query
Command
Read Model
Write Model
Projection
Process Manager
Business Rules
Event
Event Bus
Event Store

CQRS-ES


API
Query
Command
Read Model
Write Model
Projection
Process Manager
Aggregate
Event
Event Bus
Event Store

CQRS-ES

API

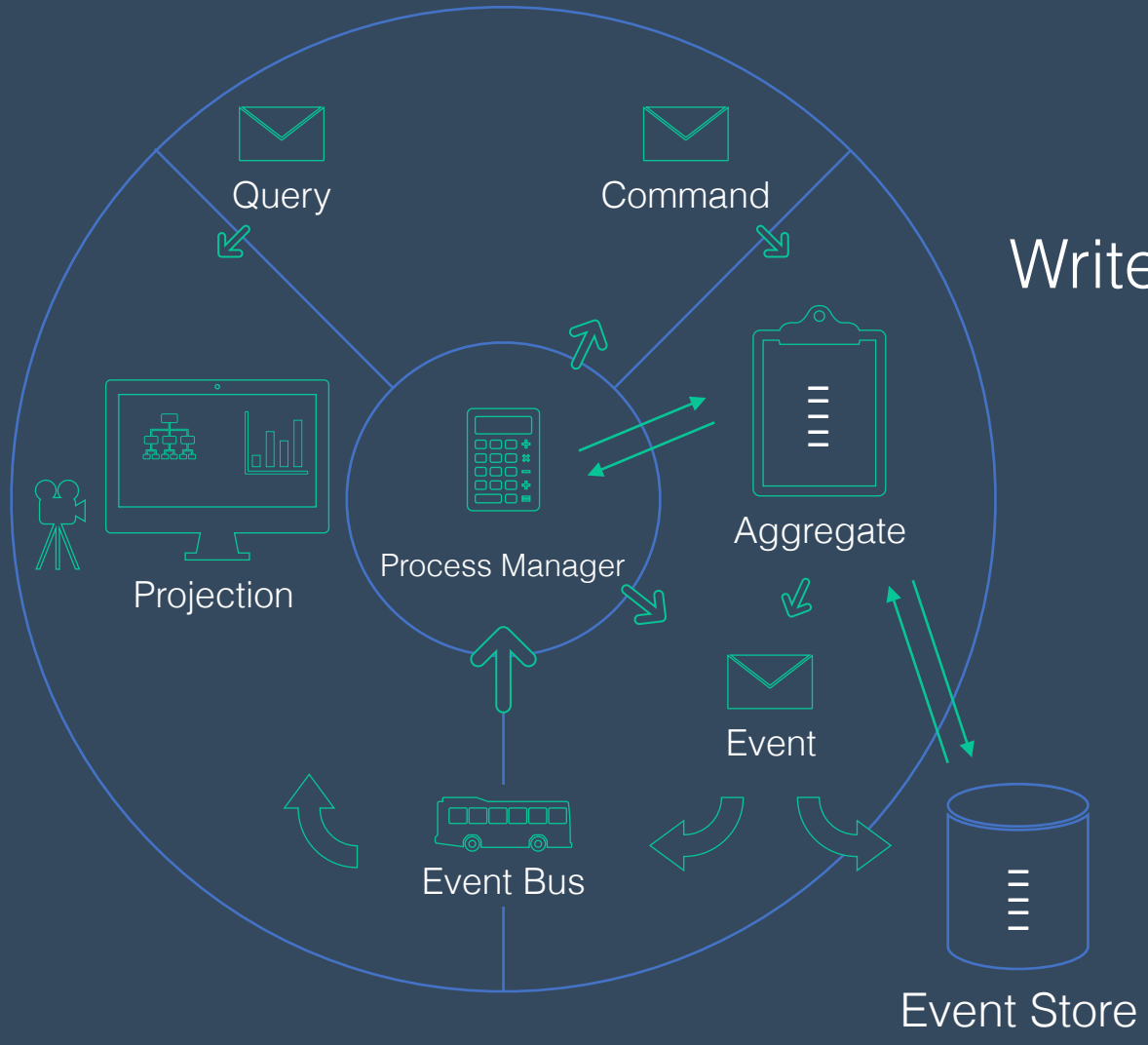

Read Model
Write Model
Query
Command
Projection
Process Manager
Aggregate
Event
Event Bus
Event Store

CQRS-ES

API

Read Model

Write Model

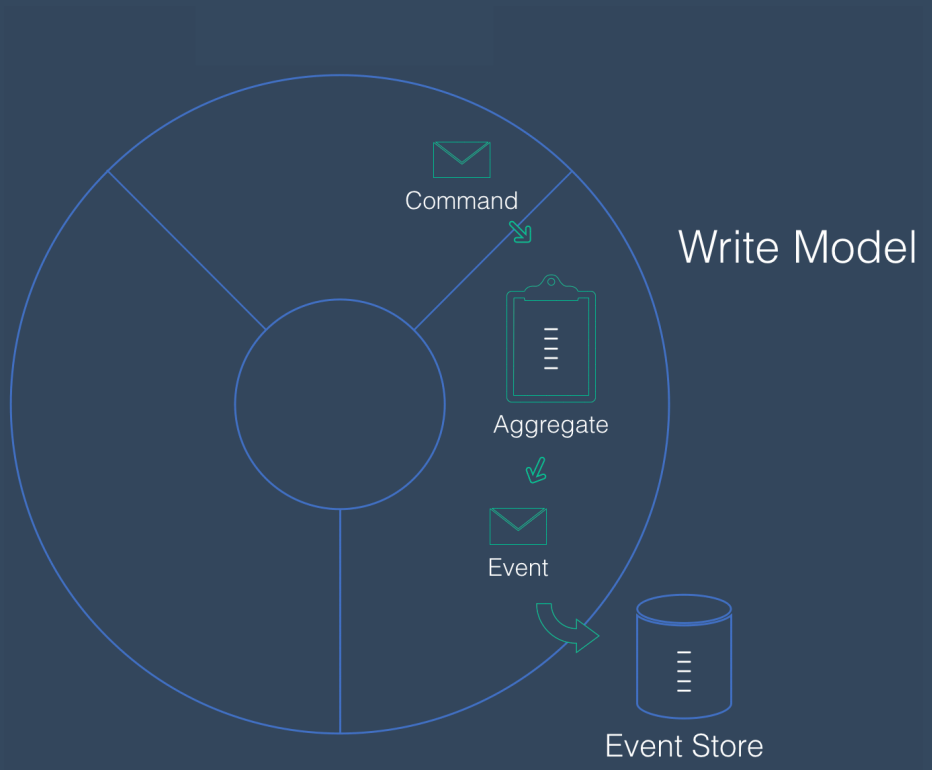


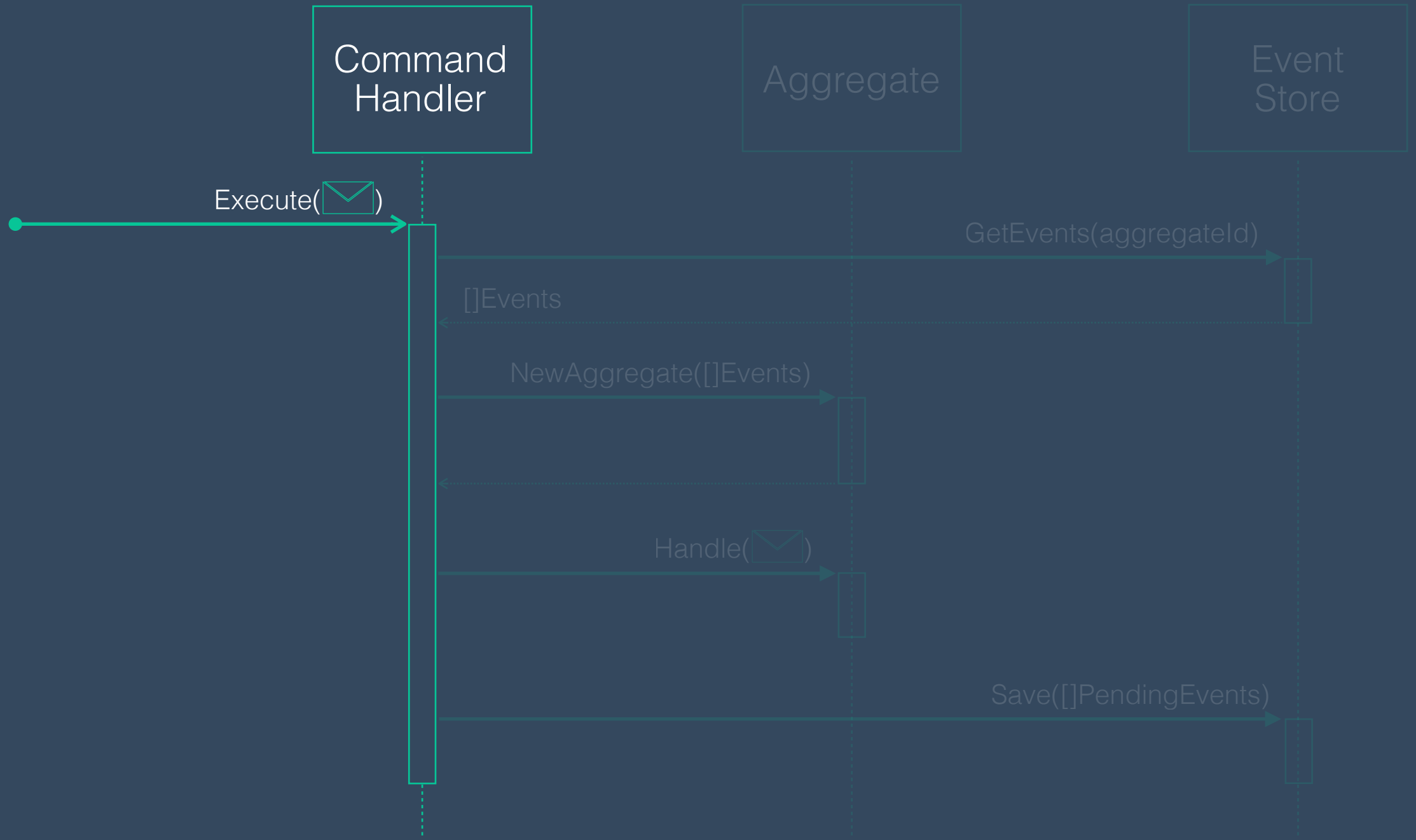
Code Examples

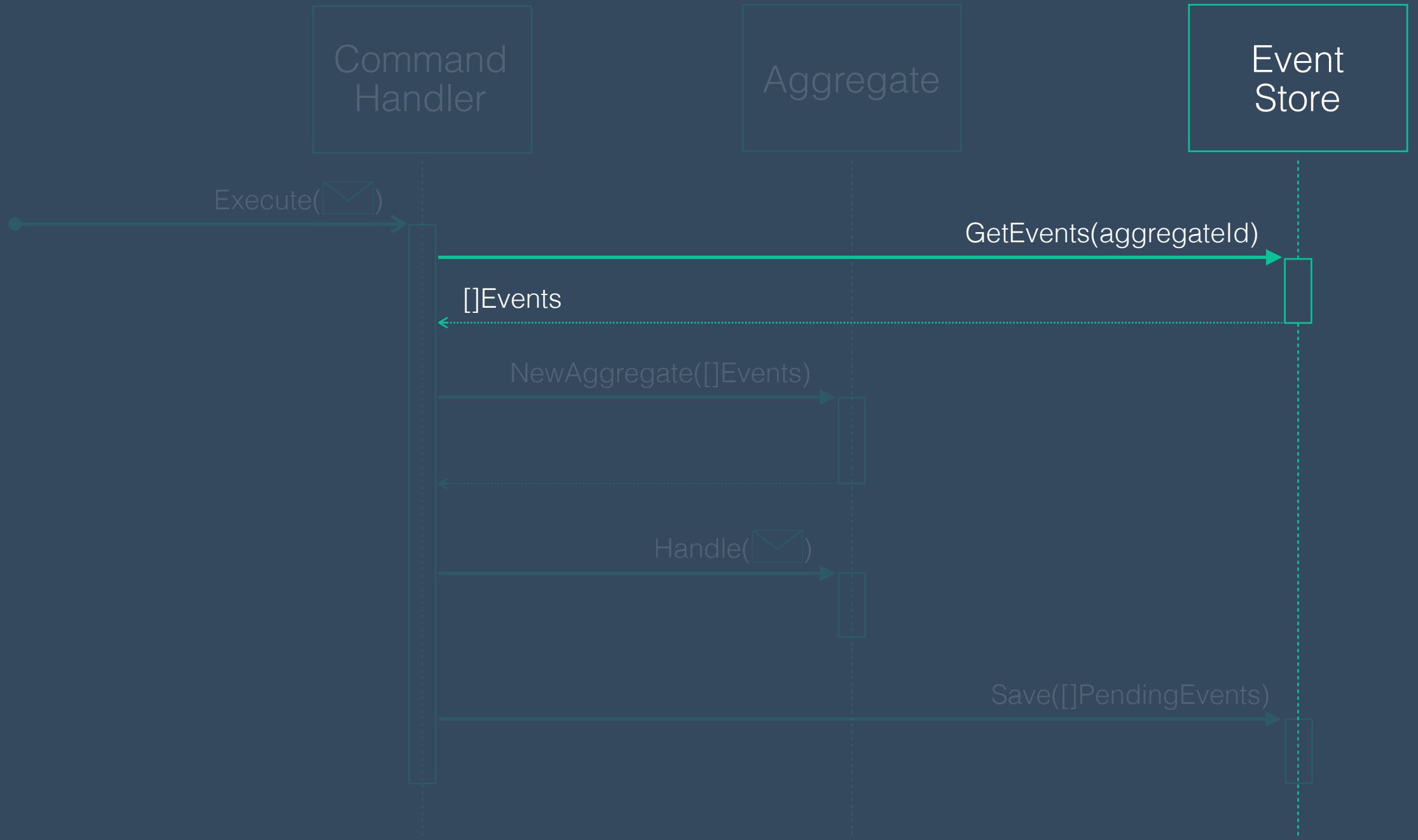
Bank Account

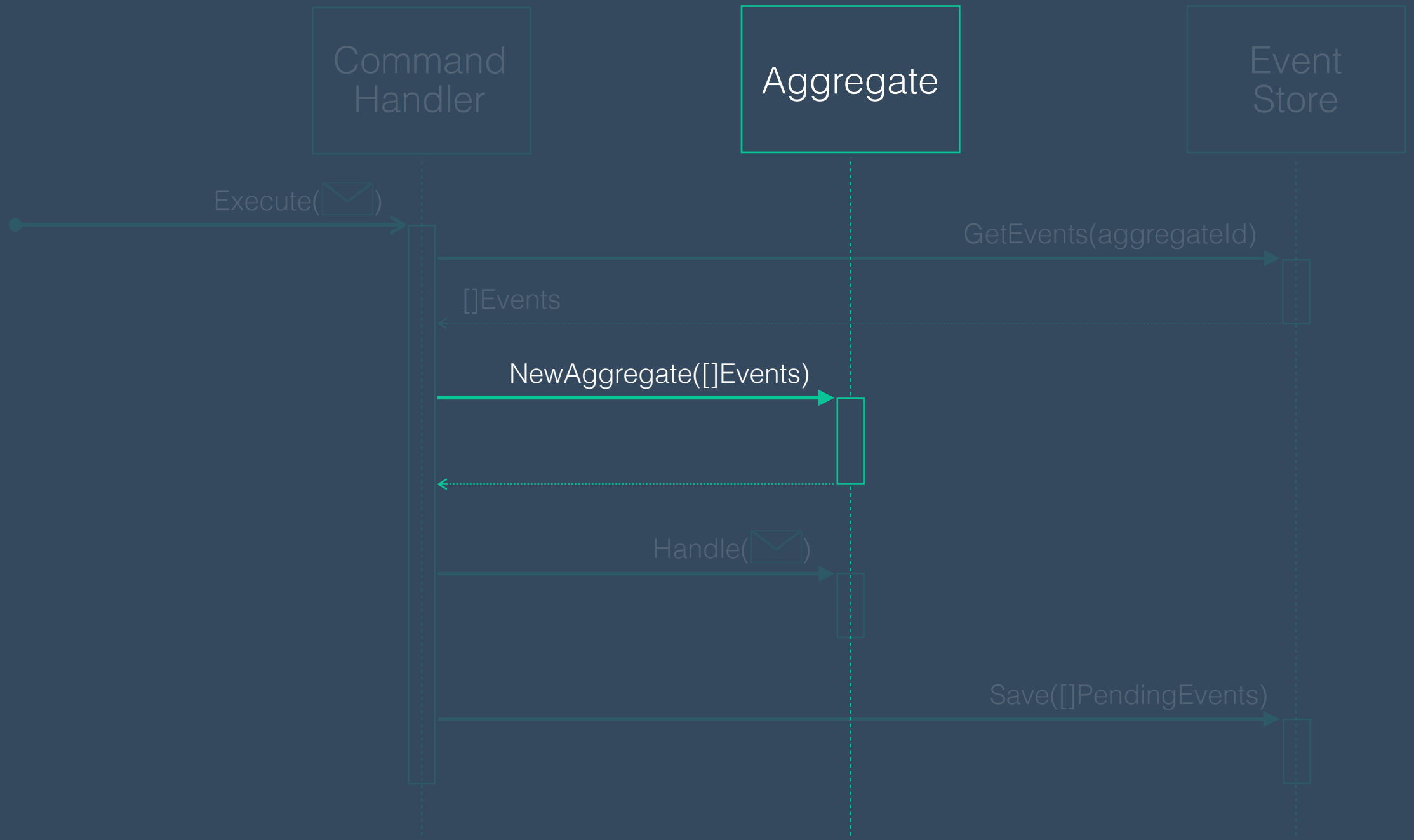
Command

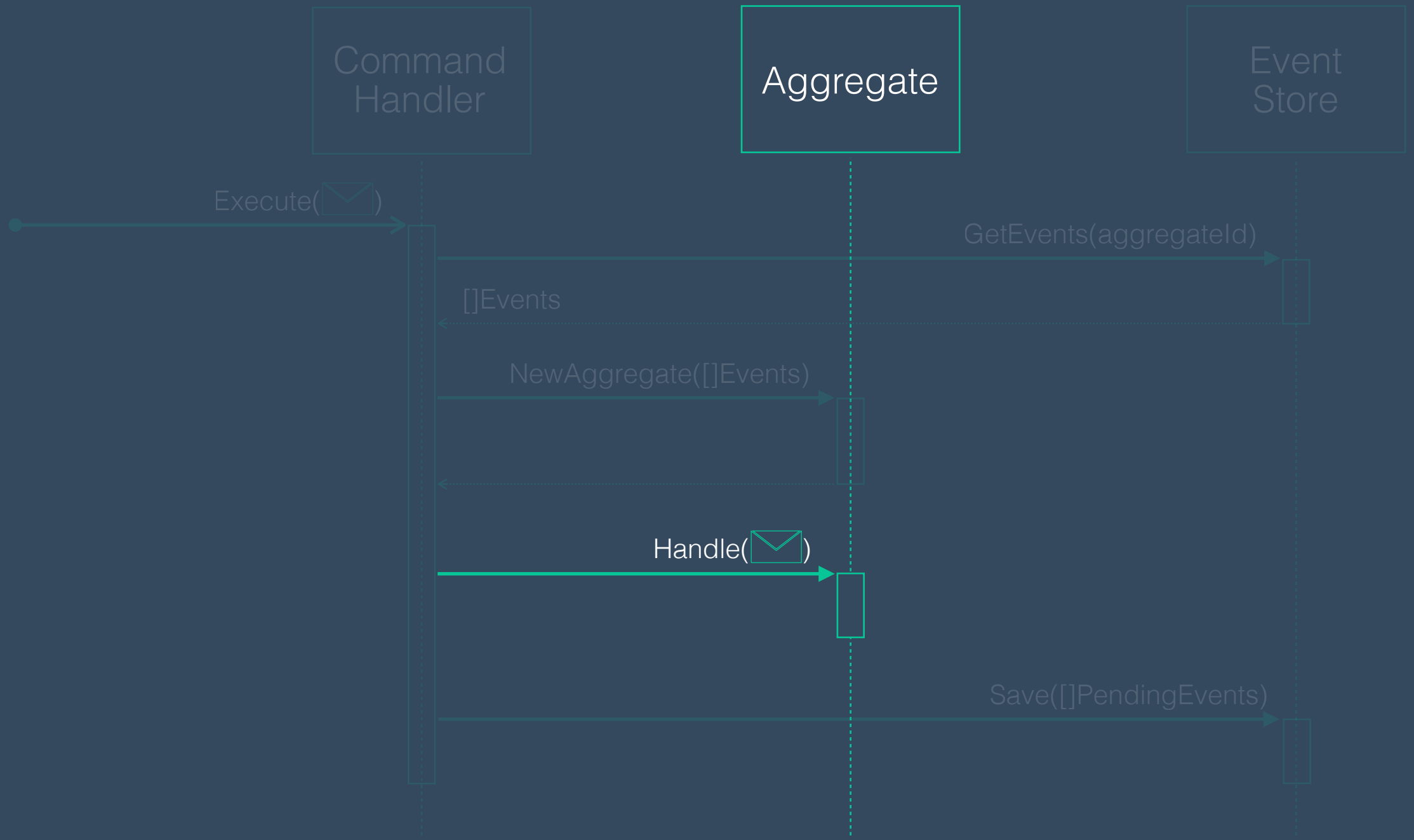
Bank Account

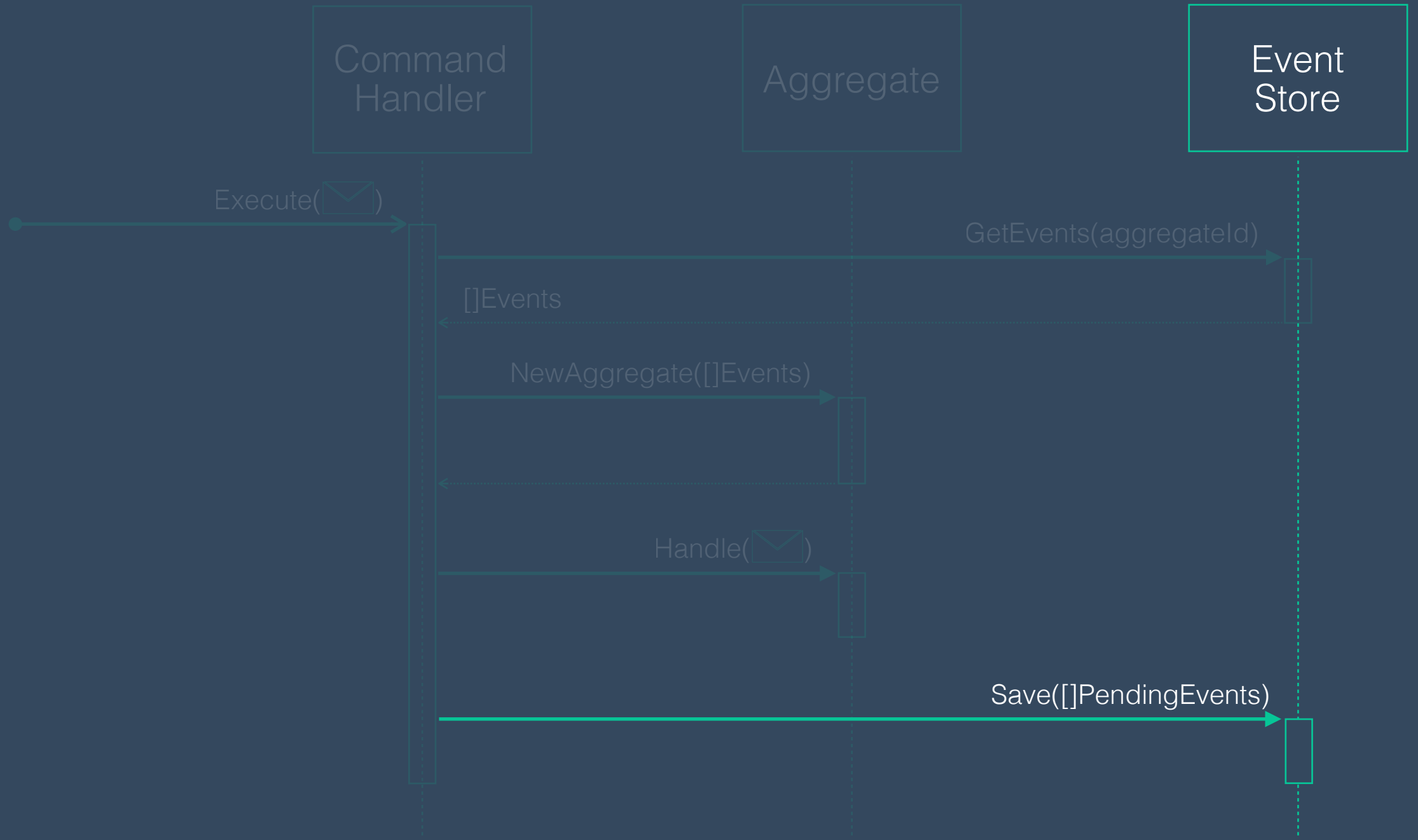


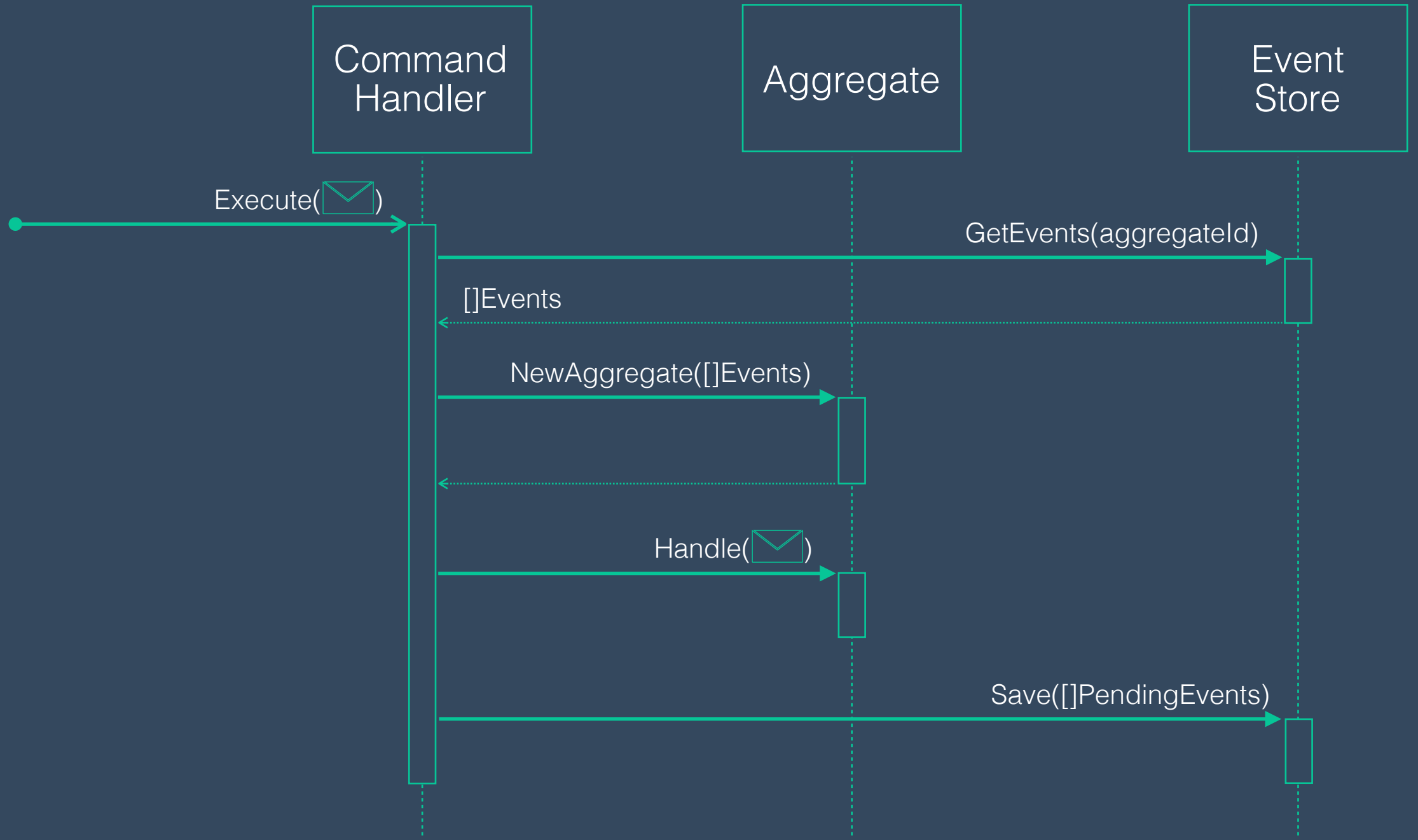

Command Handler
Aggregate
Event Store
Execute( )
GetEvents(aggregateId)
[]Events
NewAggregate([]Events)
Handle( )
Save([]PendingEvents)


Command Handler
Aggregate
Event Store
Execute(
)
GetEvents(aggregateId)
[]Events
NewAggregate([]Events)
Handle(
)
Save([]PendingEvents)


Command Handler
Aggregate
Event Store
Execute( )
GetEvents(aggregateId)
[]Events
NewAggregate([]Events)
Handle( )
Save([]PendingEvents)


Command Handler
Aggregate
Event Store
Execute( )
GetEvents(aggregateId)
[]Events
NewAggregate([]Events)
Handle( )
Save([]PendingEvents)


Command Handler
Aggregate
Event Store
Execute(
)
GetEvents(aggregateId)
[]Events
NewAggregate([]Events)
Handle(
)
Save([]PendingEvents)


Command Handler
Aggregate
Event Store
Execute( )
GetEvents(aggregateId)
[]Events
NewAggregate([]Events)
Handle( )
Save([]PendingEvents)

Just Do It!

UNIT TEST

```go
func TestOpenAccount_Emits_AccountWasOpened(t *testing.T) {
    // Given
    app := NewTestApp()

    // When
    app.Execute(
        bank.OpenAccount{
            AccountId: accountId,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.AccountWasOpened{
            AccountId: accountId,
        })
}
```

UNIT TEST

```go
func TestOpenAccount_Emits_AccountWasOpened(t *testing.T) {
    // Given
    app := NewTestApp()

    // When
    app.Execute(
        bank.OpenAccount{
            AccountId: accountId,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.AccountWasOpened{
            AccountId: accountId,
        })
}
```

UNIT TEST

```go
func TestOpenAccount_Emits_AccountWasOpened(t *testing.T) {
    // Given
    app := NewTestApp()

    // When
    app.Execute(
        bank.OpenAccount{
            AccountId: accountId,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.AccountWasOpened{
            AccountId: accountId,
        })
}
```

UNIT TEST

```go
func TestOpenAccount_Emits_AccountWasOpened(t *testing.T) {
    // Given
    app := NewTestApp()

    // When
    app.Execute(
        bank.OpenAccount{
            AccountId: accountId,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.AccountWasOpened{
            AccountId: accountId,
        })
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}

func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}

func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}

func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```

AGGREGATE

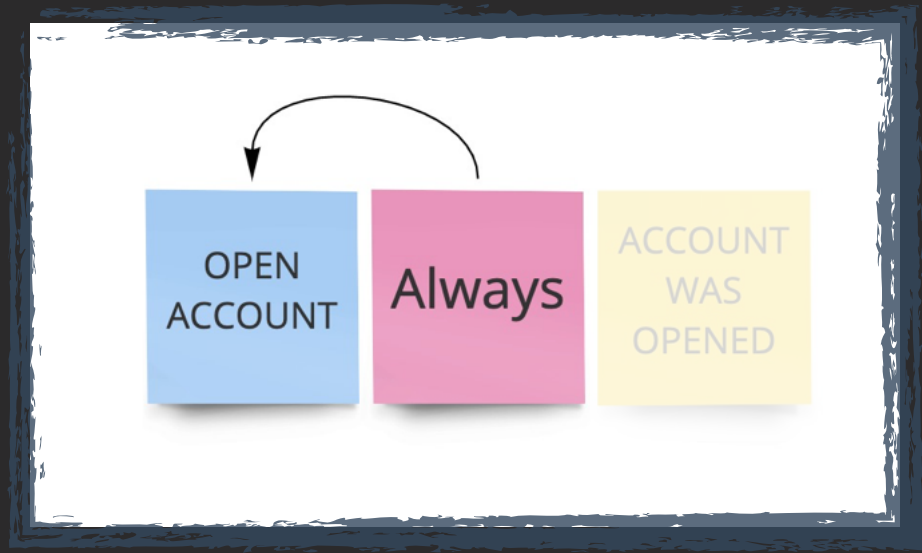
```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.emitEvent(AccountWasOpened{
            AccountId: c.AccountId,
        })

    }
}
```

AGGREGATE

```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.emitEvent(AccountWasOpened{
            AccountId: c.AccountId,
        })

    }
}
```

AGGREGATE

```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.emitEvent(AccountWasOpened{
            AccountId: c.AccountId,
        })

    }
}
```



AGGREGATE

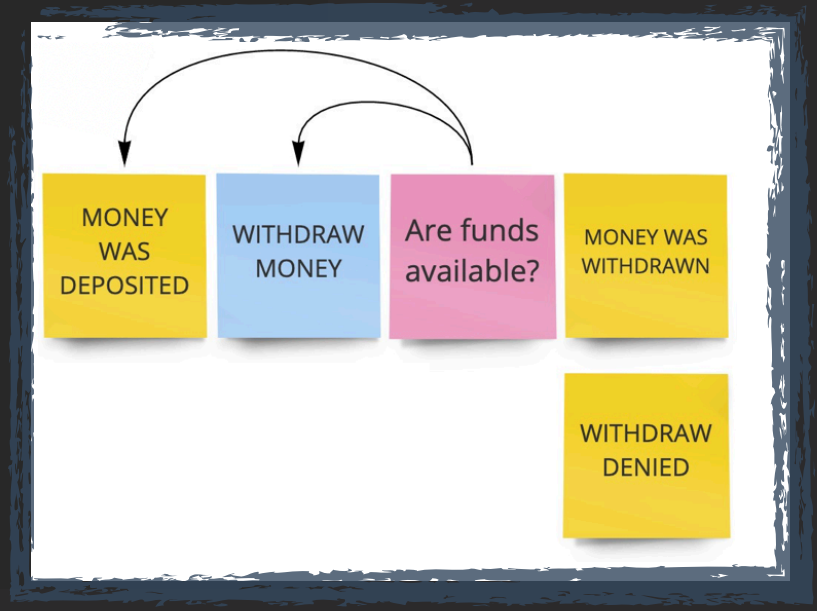
```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.emitEvent(AccountWasOpened{
            AccountId: c.AccountId,
        })

    }
}
```

COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}

func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```

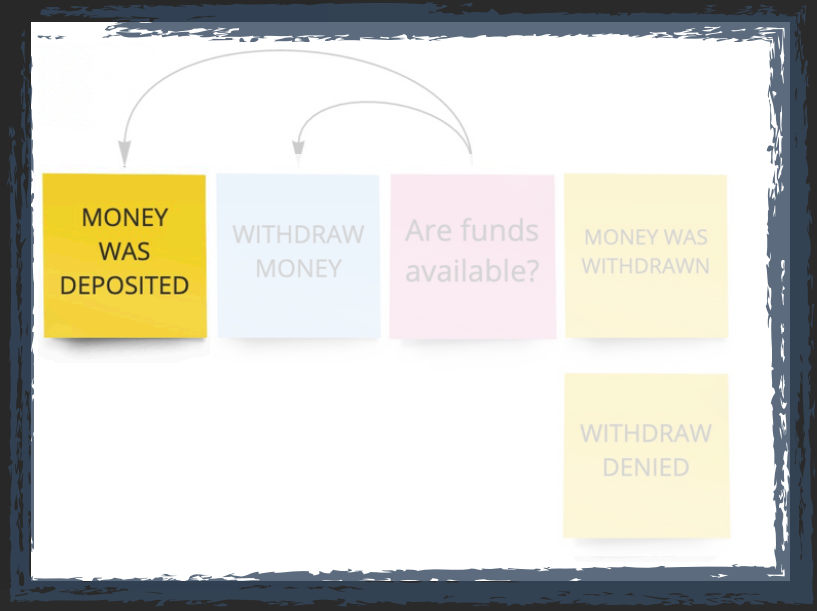
UNIT TEST

```
func TestWithdrawMoney_WhenFundsAreAvailable_Emits_MoneyWasWithdrawn(t *testing.T) {
    // Given
    app := NewTestApp()
    app.AcceptEvents(
        aggregateId,
        bank.AccountWasOpened{
            AccountId: accountId,
        },
        bank.MoneyWasDeposited{
            AccountId: accountId,
            Amount:    100,
        })

    // When
    app.Execute(
        bank.WithdrawMoney{
            AccountId: accountId,
            Amount:    75,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.MoneyWasWithdrawn{
            AccountId:  accountId,
            Amount:     75,
            NewBalance: 25,
        })
}
```



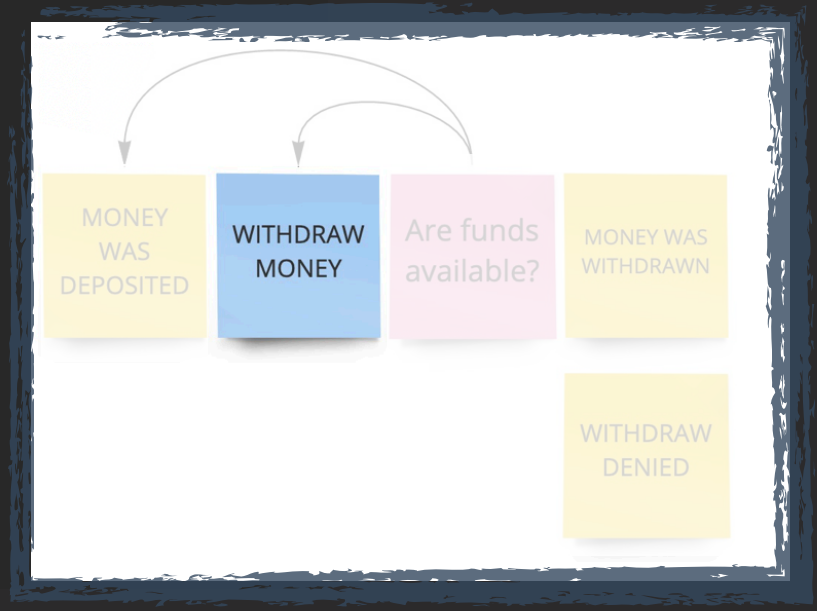
UNIT TEST

```
func TestWithdrawMoney_WhenFundsAreAvailable_Emits_MoneyWasWithdrawn(t *testing.T) {
    // Given
    app := NewTestApp()
    app.AcceptEvents(
        aggregateId,
        bank.AccountWasOpened{
            AccountId: accountId,
        },
        bank.MoneyWasDeposited{
            AccountId: accountId,
            Amount:    100,
        })

    // When
    app.Execute(
        bank.WithdrawMoney{
            AccountId: accountId,
            Amount:    75,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.MoneyWasWithdrawn{
            AccountId:  accountId,
            Amount:     75,
            NewBalance: 25,
        })
}
```



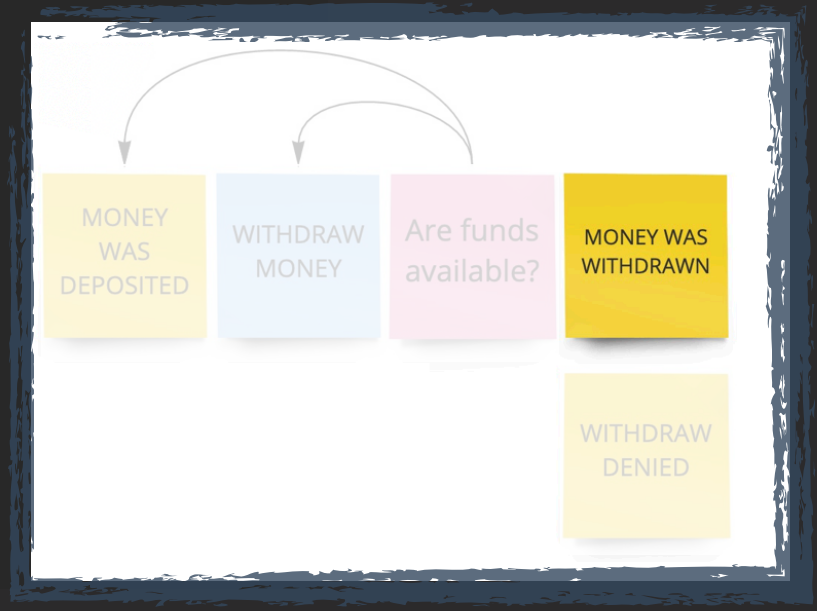
UNIT TEST

```
func TestWithdrawMoney_WhenFundsAreAvailable_Emits_MoneyWasWithdrawn(t *testing.T) {
    // Given
    app := NewTestApp()
    app.AcceptEvents(
        aggregateId,
        bank.AccountWasOpened{
            AccountId: accountId,
        },
        bank.MoneyWasDeposited{
            AccountId: accountId,
            Amount:    100,
        })

    // When
    app.Execute(
        bank.WithdrawMoney{
            AccountId: accountId,
            Amount:    75,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.MoneyWasWithdrawn{
            AccountId:  accountId,
            Amount:     75,
            NewBalance: 25,
        })
}
```



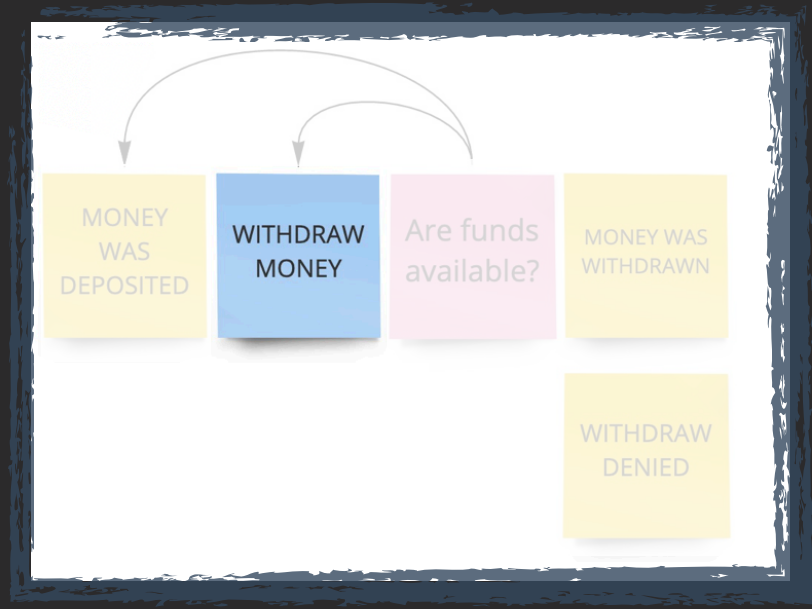
UNIT TEST

```
func TestWithdrawMoney_WhenFundsAreAvailable_Emits_MoneyWasWithdrawn(t *testing.T) {
    // Given
    app := NewTestApp()
    app.AcceptEvents(
        aggregateId,
        bank.AccountWasOpened{
            AccountId: accountId,
        },
        bank.MoneyWasDeposited{
            AccountId: accountId,
            Amount:    100,
        })

    // When
    app.Execute(
        bank.WithdrawMoney{
            AccountId: accountId,
            Amount:    75,
        })

    // Then
    ExpectEmittedEvents(t, app,
        bank.MoneyWasWithdrawn{
            AccountId:  accountId,
            Amount:     75,
            NewBalance: 25,
        })
}
```



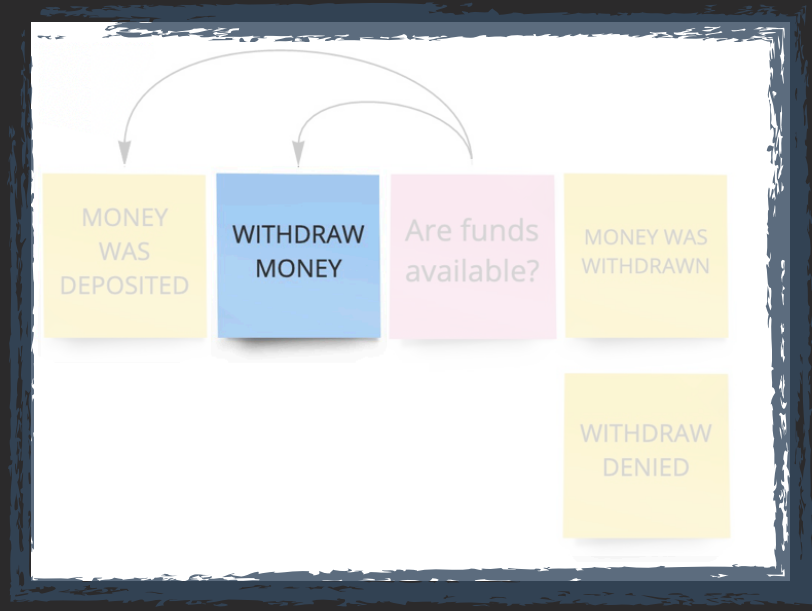
COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    case CloseAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    case DepositMoney:
        a.handleWithAccountAggregate(c.AccountId, command)

    case WithdrawMoney:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}
```



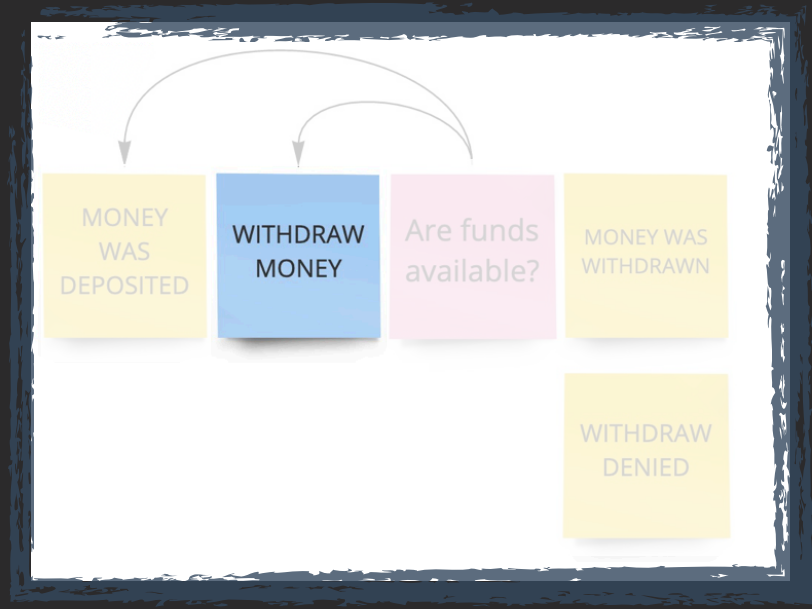
COMMAND HANDLER

```
func (a *App) Execute(command interface{}) {
    switch c := command.(type) {

    case OpenAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    case CloseAccount:
        a.handleWithAccountAggregate(c.AccountId, command)

    case DepositMoney:
        a.handleWithAccountAggregate(c.AccountId, command)

    case WithdrawMoney:
        a.handleWithAccountAggregate(c.AccountId, command)

    }
}
```



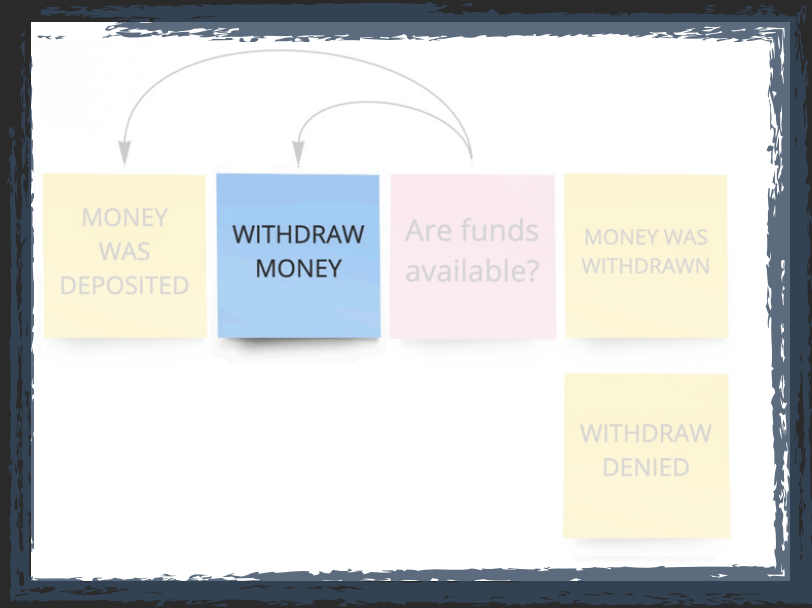
COMMAND HANDLER

```
func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```



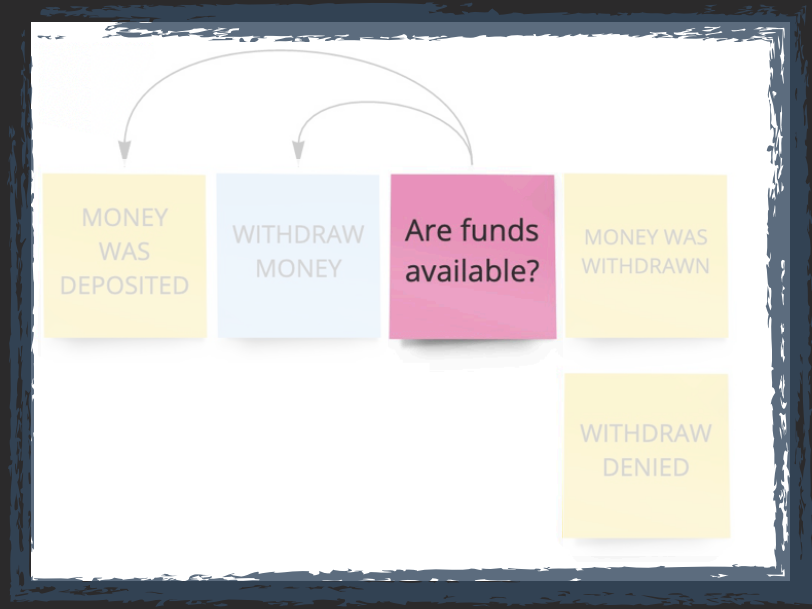
COMMAND HANDLER

```
func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```



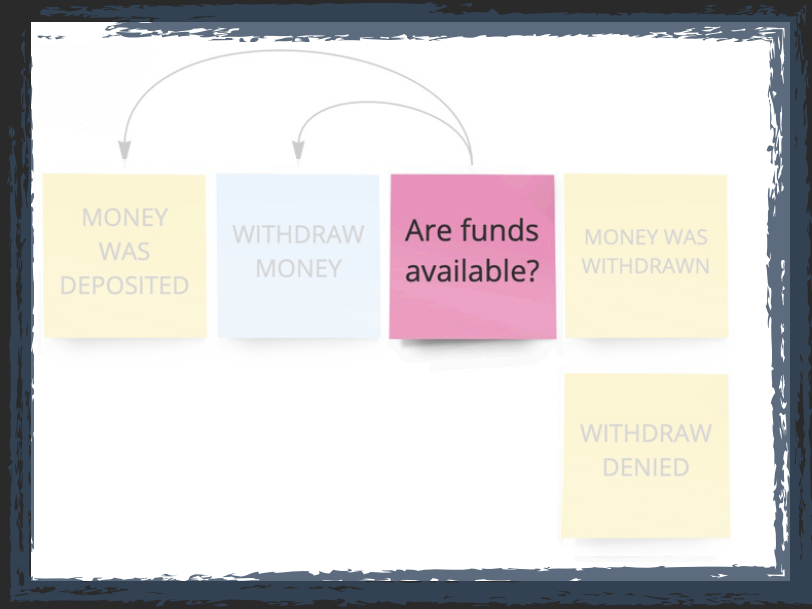
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



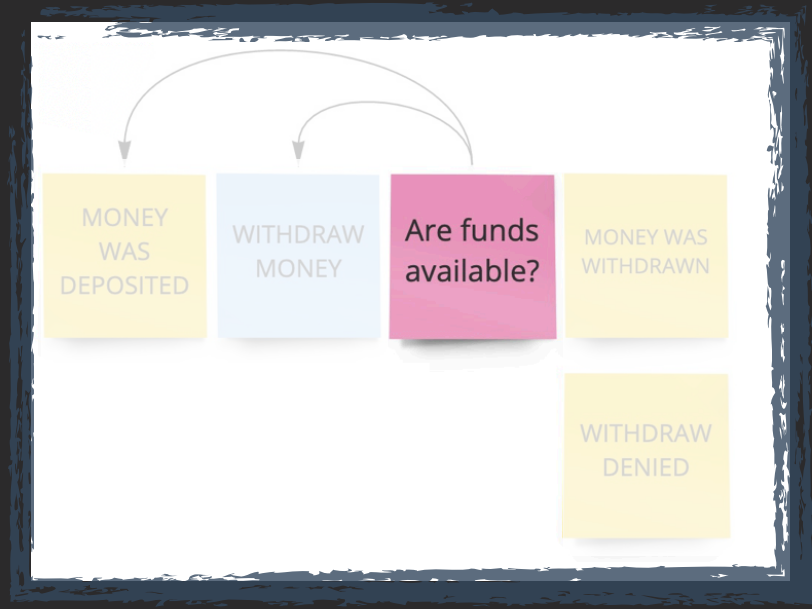
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



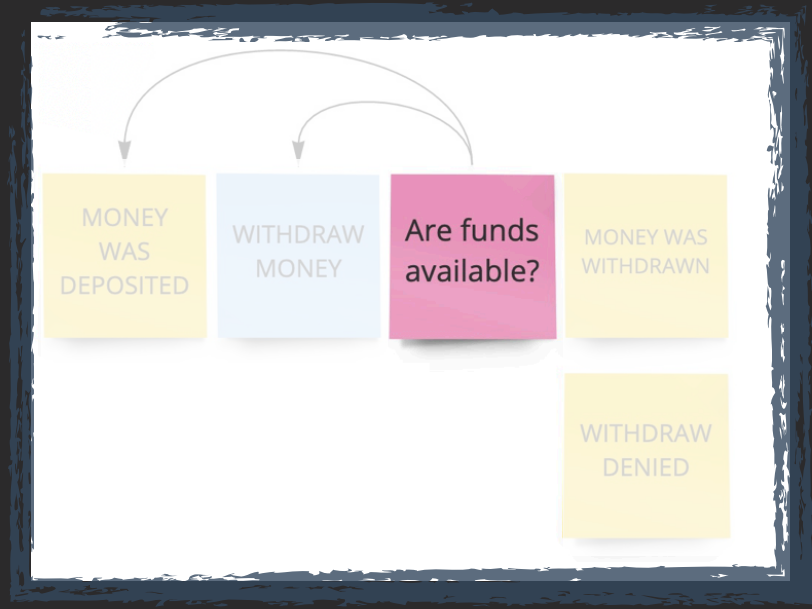
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



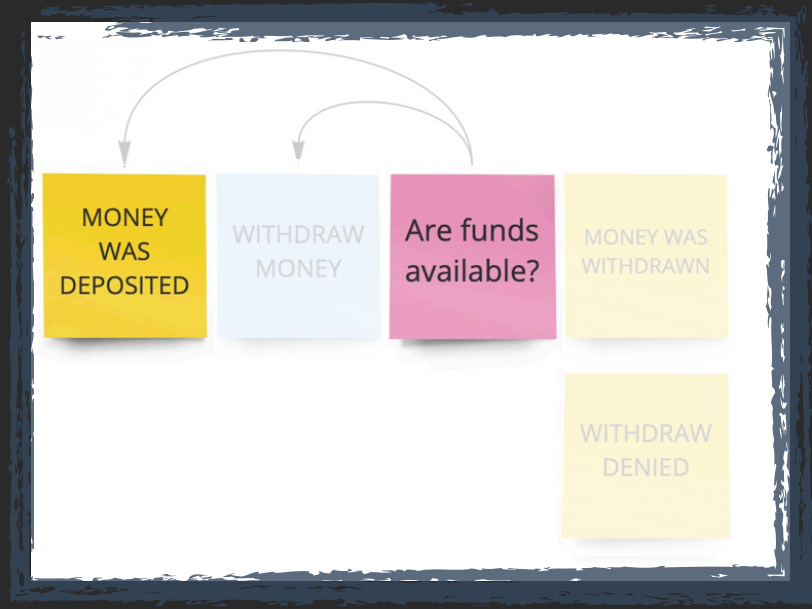
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



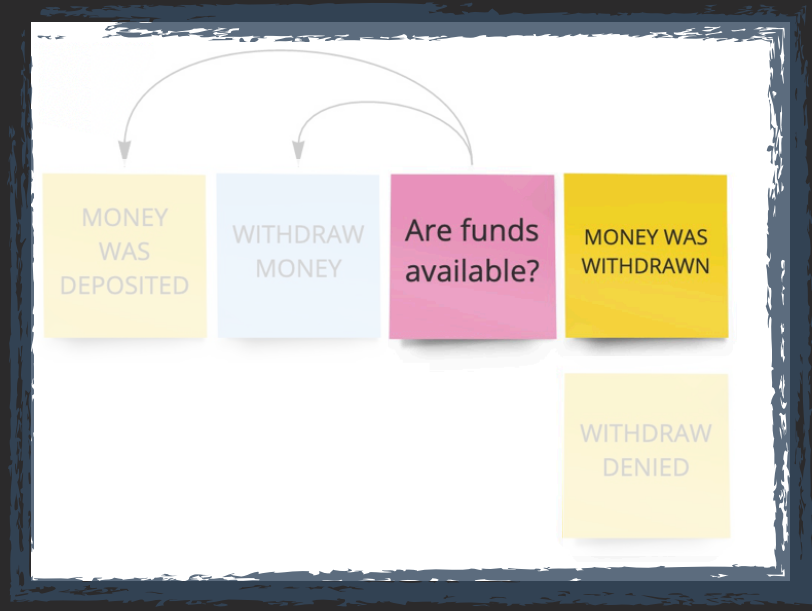
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



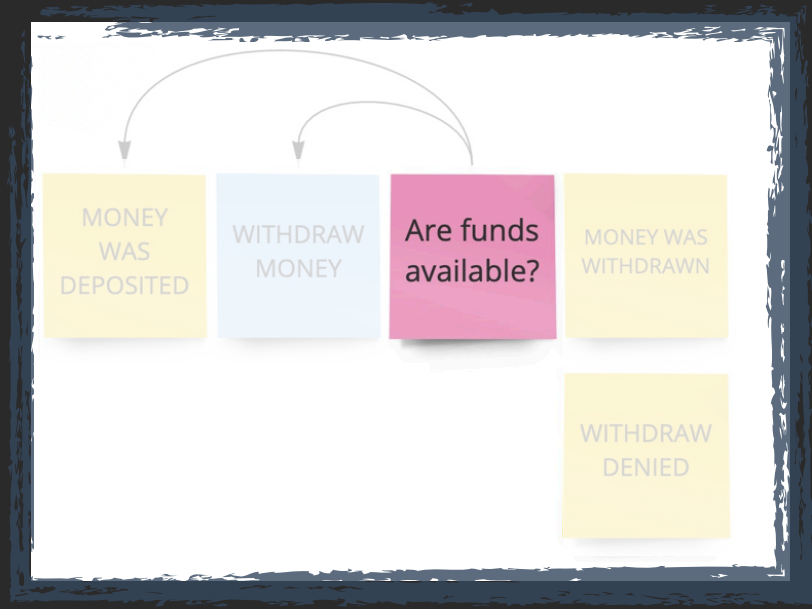
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



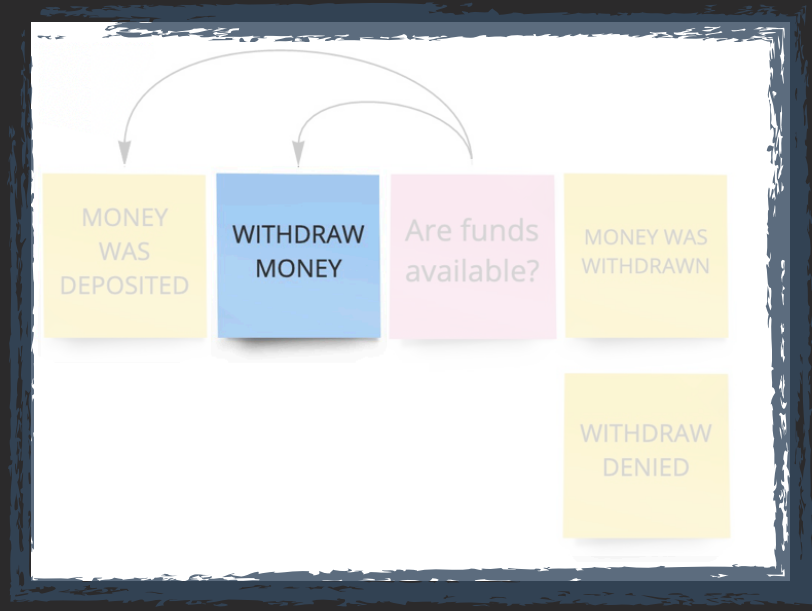
AGGREGATE

```go
func NewAccountAggregate(events []event.Event) *AccountAggregate {
    aggregate := &AccountAggregate{}

    for _, e := range events {
        aggregate.transition(e)
    }

    return aggregate
}

func (a *AccountAggregate) transition(e event.Event) {
    switch e := e.(type) {

    case AccountWasOpened:
        a.state.isOpen = true

    case MoneyWasDeposited:
        a.state.balance += e.Amount

    case MoneyWasWithdrawn:
        a.state.balance -= e.Amount

    }
}
```



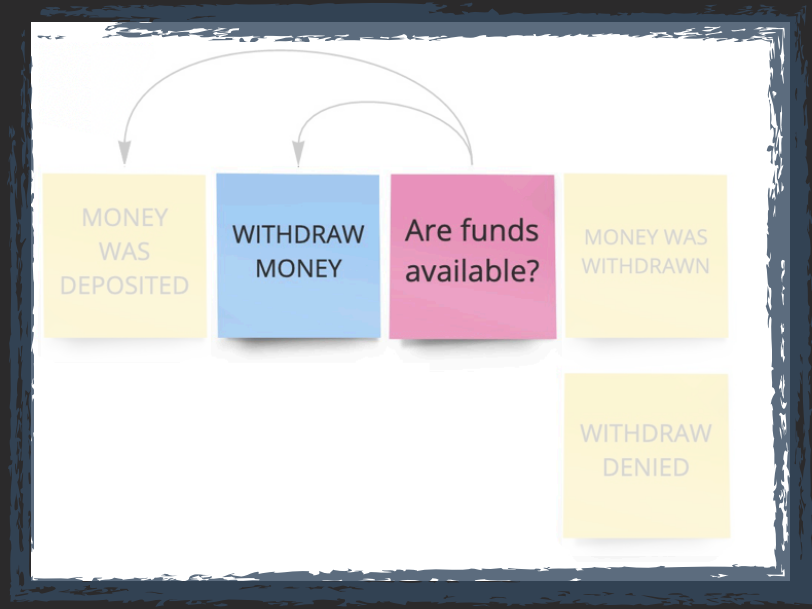
COMMAND HANDLER

```go
func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```



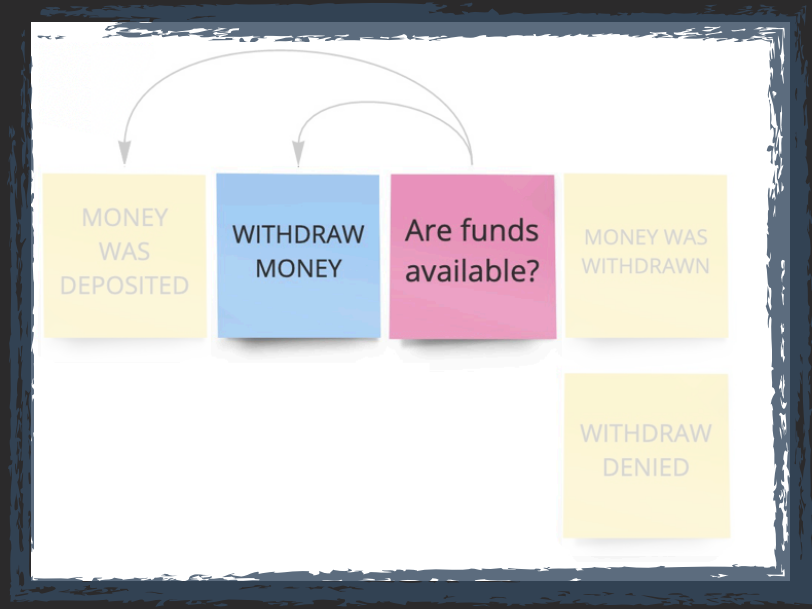
AGGREGATE

```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case WithdrawMoney:
        if a.state.balance < c.Amount {
            a.emitEvent(WithdrawDenied{
                AccountId:      c.AccountId,
                Amount:         c.Amount,
                CurrentBalance: a.state.balance,
            })
            return
        }

        a.emitEvent(MoneyWasWithdrawn{
            AccountId:  c.AccountId,
            Amount:     c.Amount,
            NewBalance: a.state.balance - c.Amount,
        })

    }
}
```



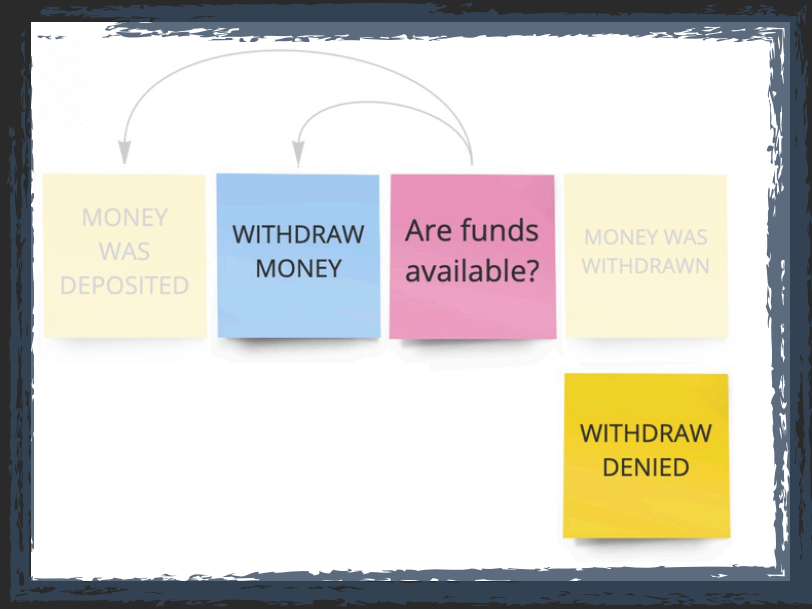
AGGREGATE

```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case WithdrawMoney:
        if a.state.balance < c.Amount {
            a.emitEvent(WithdrawDenied{
                AccountId:      c.AccountId,
                Amount:         c.Amount,
                CurrentBalance: a.state.balance,
            })
            return
        }

        a.emitEvent(MoneyWasWithdrawn{
            AccountId:  c.AccountId,
            Amount:     c.Amount,
            NewBalance: a.state.balance - c.Amount,
        })

    }
}
```



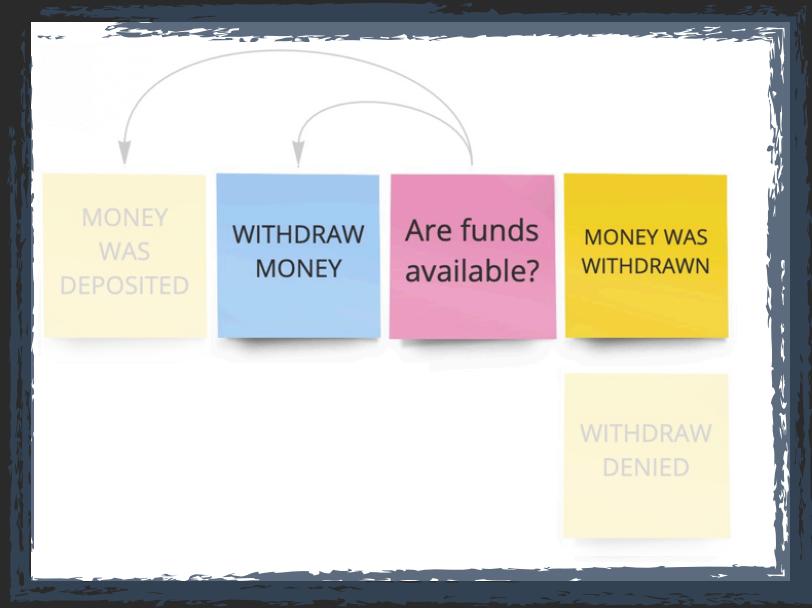
AGGREGATE

```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case WithdrawMoney:
        if a.state.balance < c.Amount {
            a.emitEvent(WithdrawDenied{
                AccountId:      c.AccountId,
                Amount:         c.Amount,
                CurrentBalance: a.state.balance,
            })
            return
        }

        a.emitEvent(MoneyWasWithdrawn{
            AccountId:  c.AccountId,
            Amount:     c.Amount,
            NewBalance: a.state.balance - c.Amount,
        })

    }
}
```



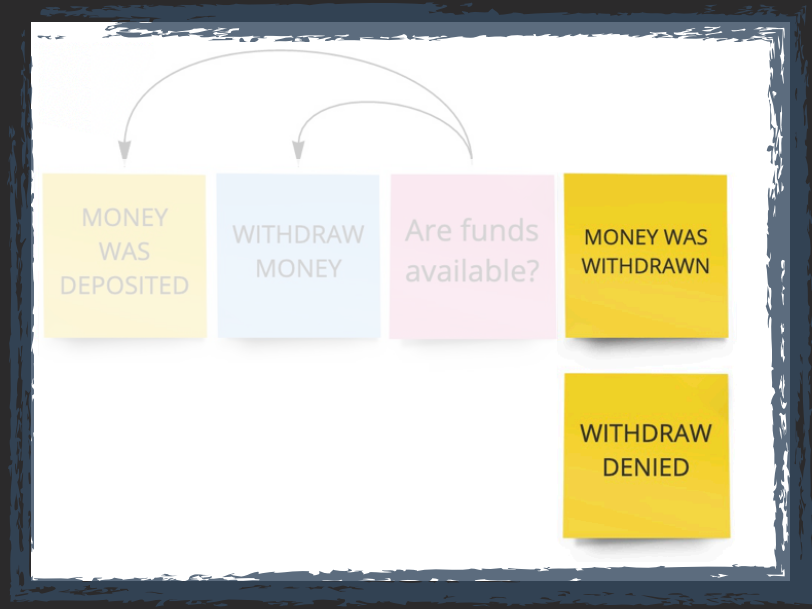
AGGREGATE

```
func (a *AccountAggregate) Handle(command interface{}) {
    switch c := command.(type) {

    case WithdrawMoney:
        if a.state.balance < c.Amount {
            a.emitEvent(WithdrawDenied{
                AccountId:      c.AccountId,
                Amount:         c.Amount,
                CurrentBalance: a.state.balance,
            })
            return
        }

        a.emitEvent(MoneyWasWithdrawn{
            AccountId:  c.AccountId,
            Amount:     c.Amount,
            NewBalance: a.state.balance - c.Amount,
        })

    }
}
```



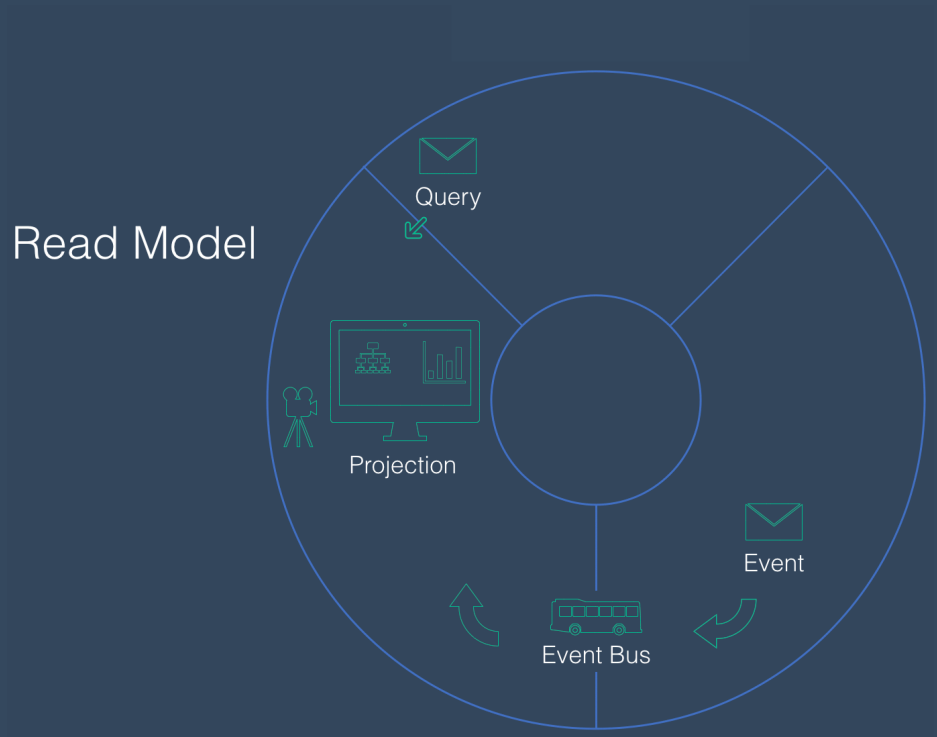
COMMAND HANDLER

```
func (a *App) handleWithAccountAggregate(aggregateId string, command interface{}) {
    events := a.eventStore.Events(aggregateId)
    account := NewAccountAggregate(events)
    account.Handle(command)
    a.eventStore.Save(aggregateId, account.PendingEvents...)
}
```



Projection

Bank Account



UNIT TEST

```go
func TestCountEvents_CalculatesTotalEvents(t *testing.T) {
    // Given
    countEvents := projection.NewCountEvents()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(countEvents)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 75},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 25},
        bank.AccountWasClosed{AccountId: "A"},
    )

    // Then
    assert.Equal(t,  expected: 5, countEvents.TotalEvents)
}
```

UNIT TEST

```go
func TestCountEvents_CalculatesTotalEvents(t *testing.T) {
    // Given
    countEvents := projection.NewCountEvents()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(countEvents)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 75},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 25},
        bank.AccountWasClosed{AccountId: "A"},
    )

    // Then
    assert.Equal(t,  expected: 5, countEvents.TotalEvents)
}
```

UNIT TEST

```go
func TestCountEvents_CalculatesTotalEvents(t *testing.T) {
    // Given
    countEvents := projection.NewCountEvents()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(countEvents)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 75},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 25},
        bank.AccountWasClosed{AccountId: "A"},
    )

    // Then
    assert.Equal(t,  expected: 5, countEvents.TotalEvents)
}
```

UNIT TEST

```go
func TestCountEvents_CalculatesTotalEvents(t *testing.T) {
    // Given
    countEvents := projection.NewCountEvents()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(countEvents)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 75},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 25},
        bank.AccountWasClosed{AccountId: "A"},
    )

    // Then
    assert.Equal(t, expected: 5, countEvents.TotalEvents)
}
```

PROJECTION

```
type countEvents struct {
    TotalEvents int
}

func (c *countEvents) Accept(event event.Event) {
    c.TotalEvents++
}
```

PROJECTION

```
type countEvents struct {
    TotalEvents int
}

func (c *countEvents) Accept(event event.Event) {
    c.TotalEvents++
}
```

PROJECTION

```go
func TestAccountFunds_CalculatesTotalFundsAndAccountBalances(t *testing.T) {
    // Given
    accountFunds := projection.NewAccountFunds()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(accountFunds)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 50},
        bank.AccountWasOpened{AccountId: "B"},
        bank.MoneyWasDeposited{AccountId: "B", Amount: 25},
    )

    // Then
    assert.Equal(t, expected: 75, accountFunds.TotalFunds)
    assert.Equal(t, expected: 50, accountFunds.AccountBalance["A"])
    assert.Equal(t, expected: 25, accountFunds.AccountBalance["B"])
}
```

PROJECTION

```go
func TestAccountFunds_CalculatesTotalFundsAndAccountBalances(t *testing.T) {
    // Given
    accountFunds := projection.NewAccountFunds()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(accountFunds)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 50},
        bank.AccountWasOpened{AccountId: "B"},
        bank.MoneyWasDeposited{AccountId: "B", Amount: 25},
    )

    // Then
    assert.Equal(t, expected: 75, accountFunds.TotalFunds)
    assert.Equal(t, expected: 50, accountFunds.AccountBalance["A"])
    assert.Equal(t, expected: 25, accountFunds.AccountBalance["B"])
}
```

PROJECTION

```go
func TestAccountFunds_CalculatesTotalFundsAndAccountBalances(t *testing.T) {
    // Given
    accountFunds := projection.NewAccountFunds()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(accountFunds)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 50},
        bank.AccountWasOpened{AccountId: "B"},
        bank.MoneyWasDeposited{AccountId: "B", Amount: 25},
    )

    // Then
    assert.Equal(t, expected: 75, accountFunds.TotalFunds)
    assert.Equal(t, expected: 50, accountFunds.AccountBalance["A"])
    assert.Equal(t, expected: 25, accountFunds.AccountBalance["B"])
}
```

PROJECTION

```go
func TestAccountFunds_CalculatesTotalFundsAndAccountBalances(t *testing.T) {
    // Given
    accountFunds := projection.NewAccountFunds()
    bus := event.NewInMemoryEventBus()
    bus.Subscribe(accountFunds)

    // When
    bus.Publish(
        bank.AccountWasOpened{AccountId: "A"},
        bank.MoneyWasDeposited{AccountId: "A", Amount: 100},
        bank.MoneyWasWithdrawn{AccountId: "A", Amount: 50},
        bank.AccountWasOpened{AccountId: "B"},
        bank.MoneyWasDeposited{AccountId: "B", Amount: 25},
    )

    // Then
    assert.Equal(t, expected: 75, accountFunds.TotalFunds)
    assert.Equal(t, expected: 50, accountFunds.AccountBalance["A"])
    assert.Equal(t, expected: 25, accountFunds.AccountBalance["B"])
}
```

PROJECTION

```
type accountFunds struct {
    TotalFunds     int
    AccountBalance map[string]int
}

func (a *accountFunds) Accept(event event.Event) {
    switch e := event.(type) {

    case bank.MoneyWasDeposited:
        a.TotalFunds += e.Amount
        a.AccountBalance[e.AccountId] += e.Amount

    case bank.MoneyWasWithdrawn:
        a.TotalFunds -= e.Amount
        a.AccountBalance[e.AccountId] -= e.Amount

    }
}
```

PROJECTION

```
type accountFunds struct {
    TotalFunds     int
    AccountBalance map[string]int
}

func (a *accountFunds) Accept(event event.Event) {
    switch e := event.(type) {

    case bank.MoneyWasDeposited:
        a.TotalFunds += e.Amount
        a.AccountBalance[e.AccountId] += e.Amount

    case bank.MoneyWasWithdrawn:
        a.TotalFunds -= e.Amount
        a.AccountBalance[e.AccountId] -= e.Amount

    }
}
```

PROJECTION

```
type accountFunds struct {
    TotalFunds     int
    AccountBalance map[string]int
}

func (a *accountFunds) Accept(event event.Event) {
    switch e := event.(type) {

    case bank.MoneyWasDeposited:
        a.TotalFunds += e.Amount
        a.AccountBalance[e.AccountId] += e.Amount

    case bank.MoneyWasWithdrawn:
        a.TotalFunds -= e.Amount
        a.AccountBalance[e.AccountId] -= e.Amount

    }
}
```

PROJECTION

```
type accountFunds struct {
    TotalFunds     int
    AccountBalance map[string]int
}

func (a *accountFunds) Accept(event event.Event) {
    switch e := event.(type) {

    case bank.MoneyWasDeposited:
        a.TotalFunds += e.Amount
        a.AccountBalance[e.AccountId] += e.Amount

    case bank.MoneyWasWithdrawn:
        a.TotalFunds -= e.Amount
        a.AccountBalance[e.AccountId] -= e.Amount

    }
}
```

Questions?